conference

proceedings

2nd Java™ Virtual Machine Research and Technology Symposium (JVM '02)

San Francisco, California, USA
August 1–2, 2002

Sponsored by
The USENIX Association

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

USENIX 2nd Java™ Virtual Machine Research and Technology Symposium *San Francisco, California August 2002*

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
URL: http://www.usenix.org

The price is $24 for members and $30 for nonmembers.
Outside the U.S.A. and Canada, please add
$12 per copy for postage (via air printed matter).

Past USENIX JVM Proceedings

JVM '01	April 2001	Monterey, CA, USA	$24/30

ISBN 1-931971-01-3

Printed in the United States of America on 50% recycled paper, 10–15% post-consumer waste.

USENIX Association

Proceedings of the 2nd Java™ Virtual Machine Research and Technology Symposium (JVM '02)

August 1–2, 2002
San Francisco, California, USA

Symposium Organizers

Program Chair

Sam Midkiff, *IBM T.J. Watson Research Center*

Program Committee

Shameem Akhter, *Intel*
Urs Gleim, *Siemens AG*
David Hardin, *aJile Systems*
Michael Hind, *IBM T.J. Watson Research Center*
Jaejin Lee, *Michigan State University*
Bernd Mathiske, *Sun Microsystems*
Eliot Moss, *University of Massachusetts, Amherst*
Bill Pugh, *University of Maryland*
Matt Welsh, *University of California, Berkeley*
Saul Wold, *Sun Microsystems*

The USENIX Association Staff

External Reviewers

Perry Chong
David Grove

2nd Java™ Virtual Machine Research and Technology Symposium

August 1–2, 2002

San Francisco, California, USA

Message from the Symposium Chair v

Thursday, August 1

Memory Management
Session Chair: Bernd Mathiske, Sun Microsystems

Adaptive Garbage Collection for Battery-Operated Environments 1
G. Chen, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, The Pennsylvania State University; M. Wolczko, Sun Microsystems

Concurrent Remembered Set Refinement in Generational Garbage Collection 13
David Detlefs and Ross Knippel, Sun Microsystems; William D. Clinger, Northeastern University; Matthias Jacob, Princeton University

To Collect or Not to Collect? Machine Learning for Memory Management 27
Eva Andreasson, BEA/Appeal Virtual Machines; Frank Hoffmann, Royal Institute of Technology; Olof Lindholm, BEA/Appeal Virtual Machines

Java on x86
Session Chair: Saul Wold, Sun Microsystems

Optimizing Precision Overhead for x86 Processors 41
Takeshi Ogasawara, Hideaki Komatsu, and Toshio Nakatani, IBM Japan

Experiences Porting the Jikes RVM to Linux/IA32 51
Bowen Alpern, Maria Butrico, Anthony Cocchi, Julian Dolby, Stephen Fink, David Grove, and Ton Ngo, IBM T.J. Watson Research Center

JVM Architecture
Session Chair: Urs Gleim, Siemens AG

A Modular and Extensible JVM Infrastructure 65
Patrick Doyle and Tarek S. Abdelrahman, University of Toronto

Cross-Architectural Performance Portability of a Java Virtual Machine Implementation 79
Matthias Jacob, Princeton University; Keith Randall, Google, Inc.

Compilers 1
Session Chair: David Grove, IBM Research

An Empirical Study of Method In-lining for a Java Just-in-Time Compiler 91
Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani, IBM Tokyo Research Laboratory

Stress-testing Control Structures for Dynamic Dispatch in Java 105
Olivier Zendra, INRIA Lorraine–LORIA/McGill University; and Karel Driesen, McGill University

Friday, August 2

Real Time and Embedded

Session Chair: David Hardin, aJile Systems

A Just-in-Time Compiler for Memory-Constrained Low-Power Devices 119
Nik Shaylor, Sun Microsystems Laboratories

sEc: A Portable Interpreter Optimizing Technique for Embedded Java Virtual Machine 127
Venugopal K S, Geetha Manjunath, and Venkatesh Krishnan, Hewlett-Packard Laboratories

JaRTS: A Portable Implementation of Real-Time Core Extensions for Java 139
Urs Gleim, Siemens AG

Compilers 2

Session Chair: Matt Welsh, University of California, Berkeley

Targeting Dynamic Compilation for Embedded Environments 151
Michael Chen and Kunle Olukotun, Stanford University

Supporting Binary Compatibility with Static Compilation 165
Dachuan Yu, Zhong Shao, and Valery Trifonov, Yale University

A Lightweight Java Virtual Machine for a Stack-Based Microprocessor 181
Mirko Raner, PTSC

Message from the Symposium Chair

Over the past few years the Java language has experienced enormous growth. Much work has also been going on under the covers in code generators, garbage collection, threading improvements, and other performance and research areas. Historically this work has been presented here and there, scattered over various conferences and symposia. By bringing it together in one place we can develop a better overall picture of what is happening in the academic and corporate research arenas.

In the Java Virtual Machine Research and Technology Symposium we've created something new: a small but powerful symposium. We solicited around the globe for papers. From the 50 papers received, we were able to select 18 for presentation. The accepted papers come from all regions of the world, including Japan, Europe, and the Americas. We have a good mix of corporate and academic works. The symposium will also present an exciting collection of Work-in-Progress reports.

The program is the result of the hard work of many. First, we want to thank the authors who submitted papers, including those whose papers we weren't able to accept. Next, we want to thank the program committee members for their diligent labor. They read a large number of papers in a short time, and they shepherded the accepted papers. We also want to thank the external reviewers, who contributed numerous high-quality reviews. Finally, we would like to thank the USENIX staff for all of their detailed and timely help in putting the conference together.

Our thanks to each of these individuals for their contributions toward producing the outstanding JVM '02 program you see before you. It's been our privilege working with each of them.

Sam Midkiff
Program Chair

Adaptive Garbage Collection for Battery-Operated Environments *

G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin
Dept. of Computer Science and Engg.
The Pennsylvania State University
University Park, PA 16802
{ gchen, kandemir, vijay, mji }@cse.psu.edu

M. Wolczko
Sun Microsystems
Palo Alto, CA
mario@eng.sun.com

Abstract

Energy is an important constraint for battery-operated embedded Java environments. In this work, we show how the garbage collector (GC) can be tuned to reduce the energy consumption of Java applications. In particular, we show the importance of tuning the frequency of invoking GC based on object allocation and garbage creation rates to optimize leakage energy consumption. We reduce the leakage energy by exploiting the supply-gated leakage power optimization that is controlled by the GC. In this mechanism, power supply to memory banks that do not hold any useful data can be shut down. We implement a new adaptive GC mechanism within Sun's KVM that optimizes the ability to shut down more banks. An evaluation of our approach using various embedded applications shows that the adaptive garbage collection scheme is effective in reducing the system energy consumption across different hardware configurations.

1 Introduction

Java-based language environments and Java-enabled devices are fast becoming popular in the embedded computing world. Some of the primary features of Java that make it attractive for embedded and energy-sensitive environments are platform independence (write-once, run-anywhere), on-demand loading and compilation, remote update/ execution, the advanced software development features of object oriented programming, automatic garbage collection, and pointer and type safety. As more and more battery-operated embedded environments employ Java, we believe that system designers should focus on individual components of the JVM and try to make them as energy-efficient as possible [15, 7].

The amount of energy consumed by a device determines the duration of battery life between recharges. The main sources of energy consumption in current CMOS-technology based processors are dynamic and leakage energy [1]. Dynamic energy is consumed whenever a component is utilized. In contrast, leakage energy is consumed as long as power supply is maintained and is independent of component activity. While leakage energy used to be an insignificant part of the overall energy consumption, due to the continued shrinking of transistors and associated technology changes, leakage power has become an important portion of overall energy consumption. In fact, it is shown to be the dominant part of the chip power budget for 0.10 micron technology and below [2]. Supply gating is an effective mechanism for reducing the leakage energy that shuts down the power supply to components that are idle [2, 12]. However, supply gating has two repercussions. First, when the unit that is supply-gated is a memory element, the state of the memory is lost. Second, reactivating the power supply to the unit after a period of idleness requires a few hundreds of processor cycles. In this work, our focus is on applying supply gating mechanism to memory elements under the control of a garbage collector (GC) [9].

One of the important components of the Java virtual machine is the GC. The GC automatically determines what objects a program is no longer using, and recycles them

*This research is supported in part by NSF Awards 0073419, 0103583 & 0097998; NSF CAREER Awards 0093082 & 0093085; GSRC.

for other use. The GC is a suitable component to power off memory banks that do not hold any useful data as it has access to accurate object usage [1]. In particular, after performing a collection, the GC can turn off a bank if it does not contain any live objects. In a traditional GC implementation, garbage collection is generally invoked when there is not enough memory to allocate the new object requested. While this strategy is acceptable for a pure performance-oriented JVM implementation, in an energy-sensitive environment with a banked memory architecture, delaying collecting garbage until available free memory has run out may lead to wasted leakage energy consumption. This is because the leakage energy expended on a bank during the time it holds only garbage (dead objects) is effectively wasted. One way of reducing this wasted energy is to perform garbage collections more frequently than necessary. That is, instead of waiting until heap space is exhausted, we can invoke the GC more frequently (e.g., at regular intervals), thereby detecting the dead objects earlier and turning off power supply to the memory banks aggressively.

In this paper, we focus on an embedded Java environment and demonstrate that an adaptive garbage collection strategy, that is, a strategy that tunes the frequency of GC invocations based on object creation and garbage generation patterns, can lead to large energy savings in memory. Using KVM, Sun Microsystem's JVM designed for embedded and battery-operated environments, and a set of nine applications that are typical for hand-held devices, we also show in this paper that the energy behavior resulting from our adaptive garbage collection strategy is competitive with the best fixed allocation garbage collection. Our results also indicate that, for most of our benchmarks, as compared to the best fixed allocation garbage collection, the adaptive GC strategy incurs much less performance penalty. Based on these results, we encourage future GC implementors for embedded Java systems to explore adaptive garbage collection strategies.

The remainder of this paper is organized as follows. Section 2 presents our adaptive GC strategy. Section 3 introduces our benchmarks, presents the GC strategies compared, and discusses our simulation environment. Section 4 gives experimental results showing the effectiveness of our strategy in reducing energy consumption. We briefly discuss some related work in Section 5. Finally, Section 6 concludes the paper by summarizing our major contributions and by giving a brief outline of the planned future work on this topic.

2 Our Approach

The garbage collector in KVM is invoked when, during objection allocation, the available free heap space is insufficient to accommodate the object to be allocated. During the time between an object becomes garbage and the time it is detected to be so and collected, the object occupies space in heap memory unnecessarily. While in a pure performance-oriented environment this may not be much of a problem, in an energy-conscious environment it can cause unnecessary leakage energy consumption. For example, suppose that a bank holds only a single object that becomes garbage at time t_1. Assuming that this object is collected only at time t_2, one can see that the bank consumes leakage energy during the time period $[t_1,t_2]$ unnecessarily. Obviously, the larger the difference between these two times, the higher this wasted energy consumption. It is thus vital from the energy perspective to detect and collect garbage as soon as possible. However, previous GC algorithms that are known to collect some objects as soon as they become garbage (e.g., reference counting [9]) have other problems such as reclaiming circular structures and efficiency problems in updating counts. While many variants have been proposed to address such problems, these collectors have not been adopted widely. Then, an alternative option would be to increase the invocation frequency of widely-used garbage collectors such as the mark and sweep collectors employed the KVM. However, the potential savings obtained through more frequent garbage collection should be balanced with the additional overhead required to collect the dead objects earlier (i.e., the energy cost of garbage collection). Accesses to the heap and execution of instructions in the processor when the GC executes incur additional dynamic energy. Further, additional leakage energy is consumed due to the increased time expended due to more frequent invocation of the GC.

An important issue then is to come up with a suitable garbage collection frequency that balances the potential leakage energy savings with the additional overheads. It is not difficult to see that if we fix the GC frequency at a specific value over all applications, this value may be suitable for only some of these applications. This is because each application has a different object allocation and garbage generation pattern. Our experimental results detailed in Section 4 shows that this is really the case. Another option would be tuning the frequency of garbage collection dynamically (i.e., during execution).

In this section, we describe an adaptive garbage collection strategy that tunes the frequency of the garbage col-

[1] Memory structures are typically organized as smaller partitions called banks or sub-banks for power and performance optimizations

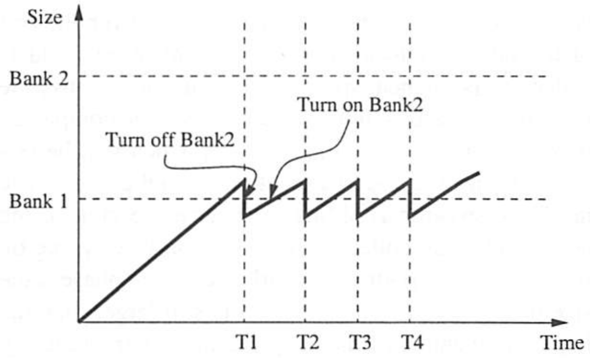


Figure 1: Garbage collector is invoked at T1, T2, T3 and T4.

lection according to the dynamic behavior of the application at runtime. A basic principle is that *whenever there is an opportunity to turn off a bank that holds only dead objects, the garbage collector should be invoked* . Simple application of this principle, however, may cause frequent turn on and turn off of the same bank (thrashing) if too few objects are collected at each collection (Figure 1). To avoid thrashing, we need an additional principle: *garbage collector should not be invoked unless a certain number of objects have become garbage* .

It should be noted, however, without actually invoking GC, it is not possible to tell exactly how many (and what size) objects have become garbage (since the last collection phase). However, due to the fact that many object allocation/deallocation patterns exhibit some regularity in one form or another [9], we can use past (history) information to estimate the size of the dead (unreachable) objects.

Let k_1 and k_2 be the object creation rate and the garbage generation rate, respectively. Assume two successive garbage collections, c_1 and c_2, that are invoked at times t_1 and t_2, respectively (Figure 2). Assume further that after c_1, the total size of the (live) objects in the heap is a_1 and that, at t_2, just before c_2, the total size (including the unreachable objects) in the heap is h_2. Finally, let us assume that after c_2 the total size of the objects in the heap (excluding the unreachable ones) is a_2. Based on this, the object creation rate during the time period $[t_1, t_2]$ is:

$$k_1 = (h_2 - a_1)/(t_2 - t_1).$$

Similarly, the garbage generation rate during $[t_1, t_2]$ can be expressed as:

$$k_2 = (h_2 - a_2)/(t_2 - t_1).$$

In our strategy, we use k_1 and k_2 to estimate the object creation and garbage generation rates after t_2 until the next collection, say c_3, is invoked. After c_3, the values of k_1 and k_2 will be updated to adapt their values to (potentially) changing heap behavior.

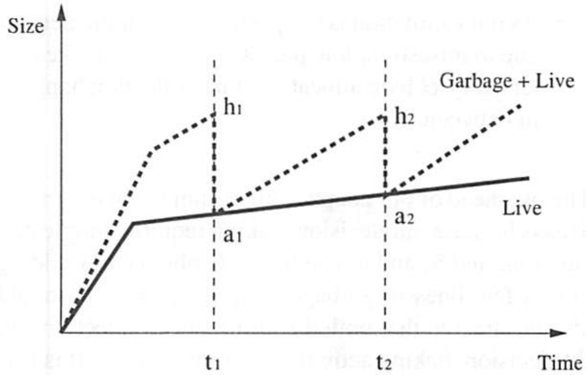


Figure 2: Garbage collection c_1 and c_2 are invoked at t_1 and t_2 respectively.

Following each allocation i that occurs at time t_i, we use the following strategy to decide whether to invoke GC or not:

- If after GC, at least two banks can be supply gated, the garbage collector should be invoked:

  $$b(h_i + s_i) - b(h_i + s_i - k_2(t_i - t_2)) \geq 2,$$

  where $b(s)$ is a function that returns the number of banks, given the total object size s; h_i is the total object size in the heap right before the allocation; and s_i is the size of the object to be allocated.

- We consider the ability to supply-gate only one bank differently, as a fast object allocation rate may require the re-activation of the supply gated unit within a short duration of time. Due to the time penalty for re-activation, it may be more beneficial to keep the power supply to the bank on if it would result in re-activation within a few allocations.

  If after the collection, one bank could be supply gated, then the space GC creates must allow the application to run for an interval no shorter than L:

  $$b(h_i + s_i) - b(h_i + s_i - k_2(t_i - t_2)) = 1 \quad \text{and}$$

  $$\begin{aligned} &b(h_i + s_i - k_2(t_i - t_2))B \\ &-(h_i + s_i - k_2(t_i - t_2)) \geq k_1 L. \end{aligned}$$

  In our implementation, L is measured in terms of the number of allocations. In the last formulation above, B is the size of a memory bank.

- As our estimation is not perfect, to limit the penalty due to misestimation, the GC should be invoked after every B byte allocation if no collection happens in between.

The overhead of our adaptive algorithm is not too much. This is because our decision making requires simple calculations and h_i and a_i can be easily obtained by adding only a few lines to garbage collector codes. It should also be stressed that both the information collection and the decision making activities consume energy. It is true that more sophisticated heuristic algorithms may give more accurate predictions. However, such algorithms can also be expected to be more complex and to require more detailed past history information, thereby potentially incurring a more significant performance (execution time) overhead.

3 Experimental Setup and Benchmarks

3.1 KVM and its Garbage Collector

Sun's K Virtual Machine (KVM) [13] is a virtual machine designed with the constraints of inexpensive embedded/mobile devices in mind. It is suitable for devices with 16/32-bit RISC/CISC microprocessors/ controllers, and with as little as 160 KB of total memory available, 128 KB of which is for the storage of the actual virtual machine and libraries themselves. The KVM technology targets smart wireless phones, mainstream personal digital assistants, pagers, and small retail payment terminals.

KVM uses two different garbage collectors, both based on mark-and-sweep [9]. These collectors differ from each other in that one of them supports compaction, whereas the other does not. A mark-and-sweep collector makes two passes over the heap. In the first pass (called the mark pass), a bit is marked for each object indicating whether the object is reachable (live). After this step, a sweep pass returns unreachable objects (garbage) to the pool of free objects. Our adaptive garbage collection strategy can be made to work with other popular collectors (e.g., a generational collector [9]) as well. In this work, we use only the compacting collector; the results with the non-compacting collector are similar, and omitted due to lack of space.

In the compacting mark-and-sweep collector used in KVM, permanent objects are distinguished from dynamic objects. A certain amount of space from the end of the heap is allocated for permanent objects and is called the permanent space. This is useful because the permanent space is not marked, swept, or compacted (since it contains permanent objects which will be referenced until the end of execution of application). The mark and sweep part of this collector is the same as the non-compacting collector. Compaction takes place on two occasions: (i) after the mark and sweep phase if the size of the object to be allocated is still larger than the largest free chunk of memory obtained after sweeping; and (ii) when the first permanent object is allocated, and, as needed, when future permanent objects are allocated. During compaction, all live objects are moved to one end of the heap. While allocating a new dynamic object, the free list is checked to see whether there is a chunk of free memory with enough space to allocate the object. If there is not, then the garbage collector is called. During garbage collection (after the sweep phase), it is checked whether the largest free chunk of memory (obtained after the sweep phase) satisfies the size to be allocated. If not, then the collector enters the compaction phase. After compaction, object allocation is attempted again. If there is still not any space, an out-of-memory exception is signaled. In the rest of this paper, this compacting mark-and-sweep collector is referred to as the *base collector* or *base* for short.

3.2 Different Versions

To see how our adaptive garbage collector performs, we compared it to a class of collectors called *k-allocation collectors* that call the GC (that is, the mark-and-sweep algorithm) once after every k object allocations. We conducted experiments with six different values of k: 10, 40, 75, 100, 250, and 500. In cases where the collection frequency of the base collector is lower than the k value used, we can expect that k-allocation collector will generate a better energy behavior. In fact, we can expect that for each application there is a k value (among the k values used in our experiments) that generates the best energy result. Consequently, for the best results, the GC frequency should be tuned to application behavior. However, this, in general, may not be possible as we may not have any information about the application's heap behavior until the runtime. Consequently, our adaptive scheme tries to detect the optimal frequency at runtime.

For each application from a given set of Java applications, a successful adaptive collector should generate at least the same energy behavior as the k-allocation collector that performs best for that application. For some applications, we can even expect the adaptive collector to generate better results than even the best k-allocation collector. This may occur, for example, when there are different phases during the course of the application's execution and each phase works best with a different garbage collection frequency. Thus, we expect the adaptive collector to be competitive with the best k-allocation collector from the energy perspective.

3.3 Architecture and Simulation Environment

In this work, we focus on a system-on-a-chip (SoC) based architecture that executes KVM applications. An SoC is an integrated circuit that contains an entire electronic system in a single sliver of silicon. Considering the fact that SoCs keep getting more and more complex, their energy demand is expected to increase significantly. This is particularly true for their memory systems as many of the applications run on SoCs are data intensive. Our SoC architecture has a CPU core and two main memory modules (in addition to some peripheral custom circuitry which is not relevant in this discussion). The processor in our SoC is a microSPARC-IIep embedded core. This core is a 100MHz, 32-bit five-stage pipelined RISC architecture that implements the SPARC architecture v8 specification. It is primarily targeted for low-cost uniprocessor applications.

In our SoC architecture, the main memory is composed of three parts. The first part contains the KVM code and associated class libraries; the second part is the heap that contains objects and method areas; and the third part holds the non-heap data that contain the runtime stack and KVM variables. Typically, the KVM code and the class libraries reside in a ROM. The ROM size we use is 128KB for the storage of the actual virtual machine and libraries themselves [4]. Since, not all libraries are used by every application, banked ROMs can provide energy savings. We activate the ROM partitions only on the first reference to the partition. A ROM partition is never disabled once it has been turned on. This helps to reduce the leakage energy consumption in memory banks not used throughout the application execution. While it may be possible to optimize the energy consumed in the ROM further using techniques such as clustering of libraries, in this study, we mainly focus only on the RAM portion of memory (SRAMs are commonly used in embedded environments as memory modules) which holds the heap. The heap (a default size of 128KB) holds both application bytecodes and application data, and is the target of our energy management strategies. An additional 32KB of SRAM is used for storing the non-heap data. We assume that the memory space is partitioned into banks and depending on whether a heap bank holds a live object or not, it can be shutdown (for saving leakage energy). In this work, each bank is assumed to be 16KB. Our objective here is to shutdown as many memory banks as possible in order to reduce leakage and dynamic energy consumption. Note that the operating system is assumed to reside in a different set of ROM banks for which no optimizations are considered here. Further, we assume a system without virtual memory support, as this is uncommon in many embedded environments [8].

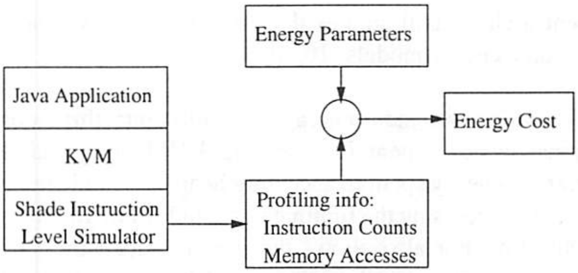

Energy Parameters
Java Application
KVM
Shade Instruction Level Simulator
Energy Cost
Profiling info: Instruction Counts Memory Accesses

Figure 3: Simulation environment.

To perform energy calculations, we built a custom simulation environment on top of the Shade [5] (SPARC instruction set simulator) tool-set (Figure 3). Shade is an instruction-set simulator and custom trace generator that simulates the SPARC V8 instruction set. Our simulator takes a KVM system with an application as input and keeps track of the energy consumption in the processor core (datapath), on-chip caches, and the on-chip SRAM and ROM memories. The datapath energy includes the energy spent during application execution and that expended during garbage collection. The energy consumed in the processor core is estimated by counting (dynamically) the number of instructions of each type and multiplying the count by the base energy consumption of the corresponding instruction. The energy consumption of the different instruction types is obtained using a customized version of a validated cycle accurate energy simulator [14]. The simulator is configured to model a five-stage pipeline similar to that of the microSPARC-IIep architecture. An analytical energy model (validated to be within 3% of actual values) similar to that proposed in [11] is used for evaluating the dynamic energy consumption of the memory elements, and a supply voltage of 1V and a threshold voltage of 0.2V. Further, we assume that the leakage energy per cycle of the entire memory is equal to the dynamic energy consumed per access. This assumption tries to capture the anticipated importance of leakage energy in 0.10 micron (and below) technologies [2]. Similar abstractions for leakage energy modeling have been employed in recent architectural studies due to the lack of validated leakage energy models [10, 3].

The memory system energy is divided into three portions: energy spent in accessing KVM code and libraries, energy spent in accessing heap data, and energy spent in accessing the runtime stack and KVM variables. Our simulator also allows the user to adjust the various parameters for these components. Energies spent in on-chip interconnects are included in the corresponding memory components. In this study, we assume that each memory bank can be in one of three states: R/W (Read/Write), Active, and Inactive. In the R/W state, the memory bank is being read or written. It consumes full dynamic energy as well as full leakage energy. In the active state, the memory bank is active (powered on) but not being read or written. It consumes no dynamic energy but full leakage energy. Finally, in the inactive state, the bank does not contain any useful data and is supply-gated. We conservatively assume that when in the inactive state a memory bank consumes 5% of its original leakage energy (as a result of supply-gating). While placing unused memory banks in inactive state may save significant leakage energy, it may also cause some performance degradation. Specifically, when a bank in the inactive state is accessed (to service a memory request), it takes 350 cycles to transition to the active state [3]. We term this time as the resynchronization time (or resynchronization penalty). It should be mentioned that both inactive state leakage energy consumption and resynchronization time are highly implementation dependent and are affected by the sizing of the driving transistors. The values that we used in this study are reasonable and conservative [3]. We also assume that the per cycle leakage energy consumed during resynchronization is the average of per cycle leakage energy consumed when the system is in the active state and that consumed when it is in the inactive state.

3.4 Benchmark Codes

To test the effectiveness of our energy saving strategy, we collected nine applications shown in Table 1. These applications represent a group of codes that are executed in energy-sensitive devices such as hand-held computers and electronic game boxes, and range from utilities such as calculator and scheduler, embedded web browser to game programs.[2] We believe that these applications represent a good mix of codes that one would expect to run under KVM-based, battery-sensitive environments.

4 Experimental Results

Unless otherwise stated, in all the results reported in this section we use an L value of 20 allocations (see Section 2) and a bank size of 16KB.

4.1 Heap Energy

Figure 4 shows the heap energy consumption for our different versions. All values shown in this graph are *normalized* with respect to the base collector. In addition to base and our adaptive collector (denoted `adpt`), we report the results for a set of k-allocation collectors with k values of 10, 40, 75, 100, 250, and 500. Above each bar is the normalized energy consumption in heap. From these results, we can make the following observations. First, as far as the k-allocation collector is concerned, we clearly see that different applications work best with different garbage collection frequencies. Second, our adaptive garbage collection strategy reduces the heap energy of the base collector in KVM significantly; the average (over all applications) heap energy saving is 28.4%. Third, for a given benchmark, the adaptive strategy is always competitive with the best k-allocation collector. For example, in `Dragon`, the adaptive strategy comes close to the 10-allocation collector (the best k-allocation collector for this benchmark). Similarly, in `Kvideo` (resp. `Kshape`), our adaptive collector generates competitive results with 75-allocation (resp. 250-allocation) collector. These results clearly show that the adaptive garbage collection is very successful in optimizing heap energy.

4.2 Overall Energy

While our objective is optimizing the energy consumption in heap, changing the frequency of garbage collections might also change the energy profile in other parts of the system such as ROM, the memory banks that hold runtime stack, and the CPU datapath. For example, a more frequent collection is expected to increase the energy spent in CPU due to garbage collection. To analyze this behavior, we give in Figure 5 the energy consumption in all components that we are interested in. We observe from these results that, for each code in our experimental suite, the adaptive strategy comes very close to the best k-allocation collector, even when one focuses on the overall energy consumption.

[2] Our applications and GC executables are publicly available from www.cse.psu.edu/∼gchen/kvmgc/

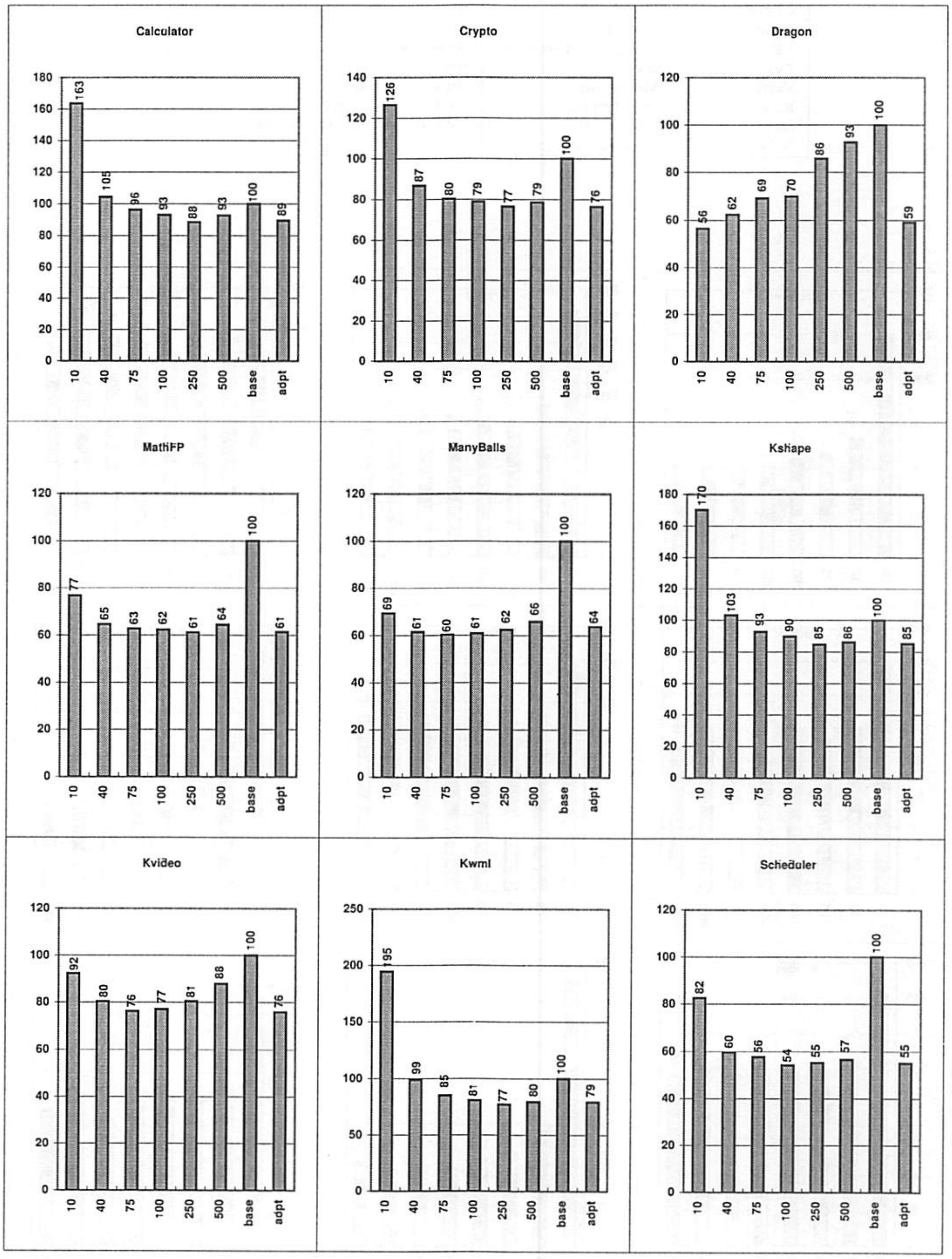


Figure 4: Normalized energy consumption (static+dynamic) in the heap memory.

Table 1: The benchmarks used in our experiments.

Application	Brief Description	Source
Calculator	Arithmetic calculator	www.cse.psu.edu/~gchen/kvmgc/
Crypto	Cryptography	www.bouncycastle.org
Dragon	Game program	comes with Sun's KVM
Kshape	Electronic map on KVM	www.jshape.com
Kvideo	KPG decoder	www.jshape.com
Kwml	WML browser	www.jshape.com
Manyballs	Game program	comes with Sun's KVM
MathFP	Integer math lib	home.rochester.rr.com/ohommes/MathFP/
Scheduler	Weekly/daily scheduler	www.cse.psu.edu/~gchen/kvmgc/

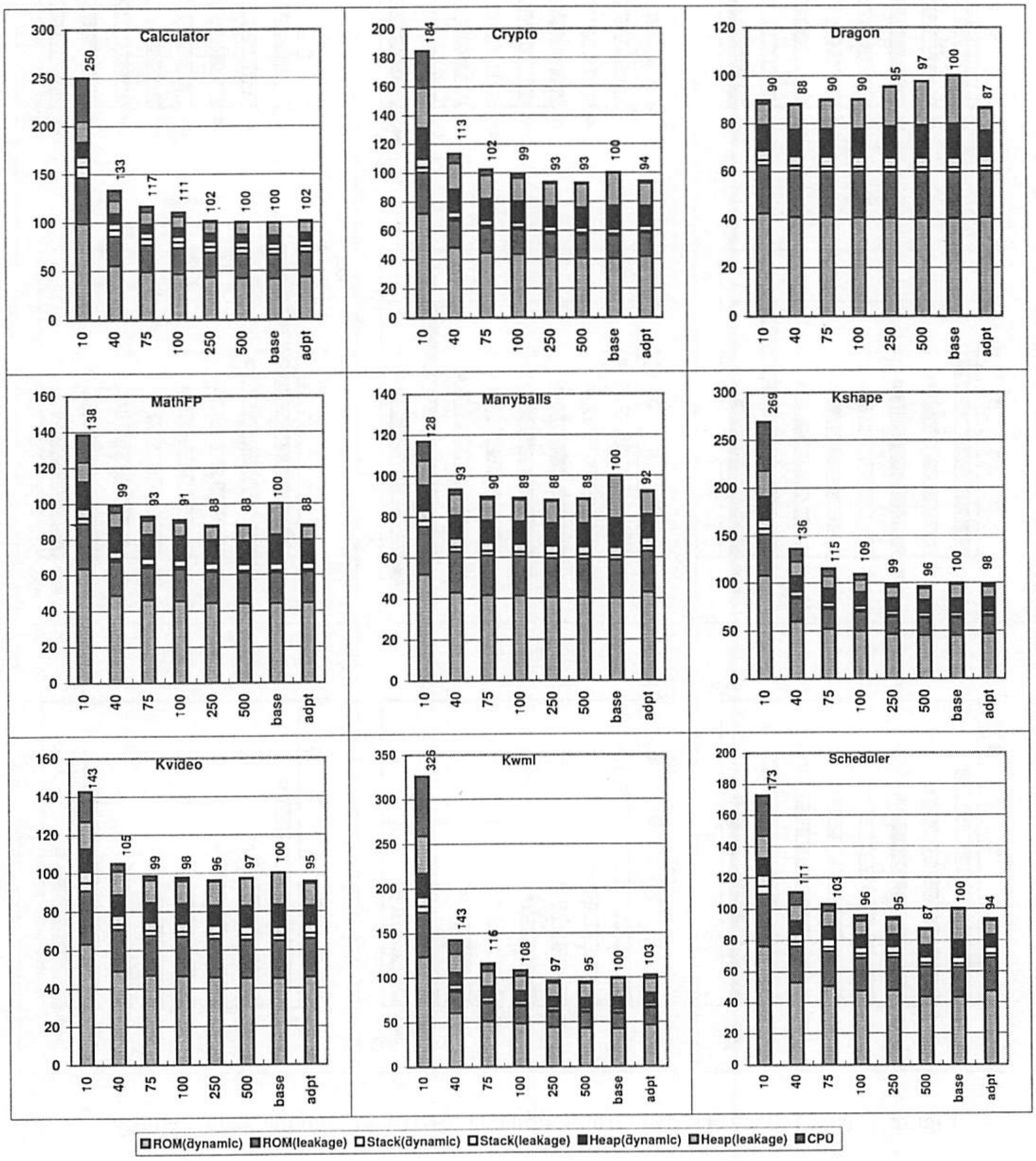


Figure 5: Normalized energy consumption breakdown.

4.3 Execution Profiles

To better understand the energy savings due to adaptive garbage collection, we give further data illustrating where the energy benefits come from. First, we give in Table 2 the number of times each version invokes garbage collection. From these results we see that the adaptive strategy calls garbage collection much more frequently than the base collector. This explain why it reduces the heap energy consumption significantly. In fact, we can see from this table that even the 500-allocation collector calls garbage collection more frequently (on average) than the base collector, indicating that the base collector is very slow in invoking the collector. As mentioned earlier however, calling garbage collection more frequently incurs an extra runtime overhead. Therefore, the garbage collector should not be invoked unless it is necessary (i.e., unless it is expected to save energy by powering off bank(s)). We see from Table 2 that the adaptive strategy calls collection less frequently than the k-allocation strategy when k is 10, 40, 75, and 100. Therefore, its runtime overhead is less than these collectors. Considering that the adaptive strategy is competitive with all these k-allocation collectors, one can see that the adaptive strategy achieves the same energy behavior as these allocators with much less performance penalty. As compared to the remaining versions, it calls collection more frequently on the average, but one application (Kwml) is responsible for this average.

Figure 6 shows the heap footprints of different versions for three applications: Calculator, Dragon, and ManyBalls. In these graphs, the x-axis corresponds to the number of allocations made during the course of execution. The y-axis, on the other hand, represents the occupied memory space in heap (by alive or dead objects). Each horizontal line on the y-axis corresponds to a bank boundary, starting from the bottom with the first bank. In these graphs, for sake of clarify, we show only a 75-allocation collector, the base collector and our adaptive collector. We can clearly observe the impact of changing the frequency of collections on the number of active power banks required. Let us first focus on Calculator. In this application, the base collector does not invoke any collection. We observe that the adaptive collector invokes collection less frequently than the 75-allocation collector (this is also true for all studied k-allocation collectors except the 500-allocation collector). When using the adaptive collector, the GC is invoked just in time before the new bank is activated. Therefore, it's energy savings are competitive with the other k-allocation collectors that invoke GC more frequently. It should also be noted that the 500-allocation collector (not shown in figure) activates collections a bit late (after the second bank is accessed). In Dragon, on the other hand, we observe a different behavior. This application experiences frequent object allocations, putting a pressure on the heap memory. Therefore, the best energy results are obtained by calling the GC frequently. Consequently, the adaptive strategy and 10-allocation collector (not shown in figure) achieve the best result. Finally, in ManyBalls, the adaptive collector strikes a balance between very frequent and very slow garbage collections.

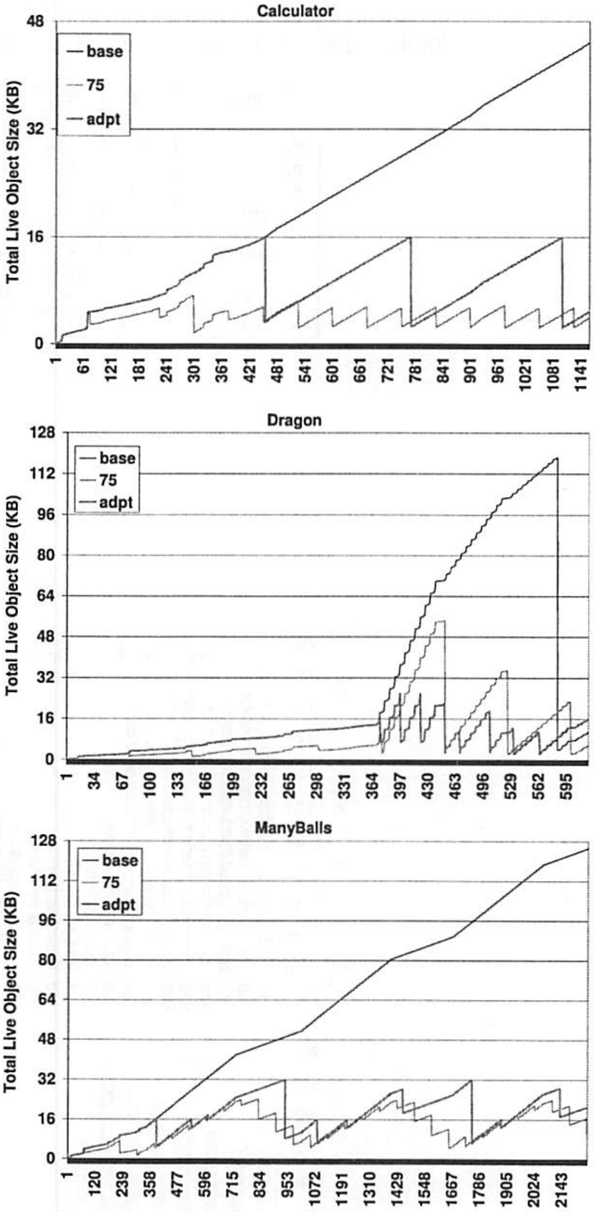
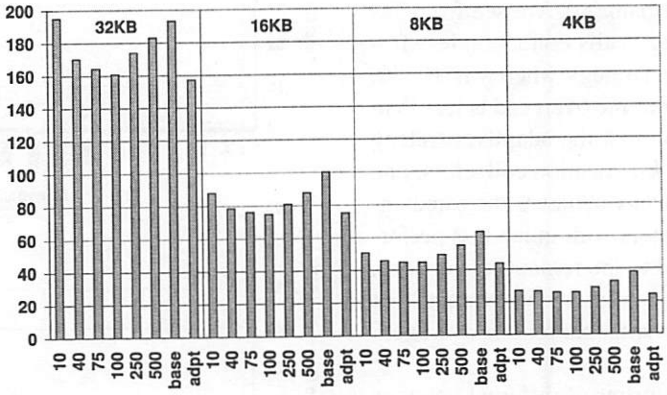


Figure 6: Execution footprints of Calculator, Dragon, and ManyBalls under different collection frequencies.

Table 3: Percentage increase in execution cycles (with respect to the base collector).

Application	10	40	75	100	250	500	adpt
Calculator	92.58	21.71	11.78	8.01	3.00	1.88	2.86
Crypto	80.57	19.89	10.34	7.73	2.73	1.08	3.29
Dragon	14.53	1.08	0.54	0.40	0.07	0.03	0.68
MathFP	47.82	11.05	5.34	3.64	0.65	-1.21	0.60
ManyBalls	25.71	6.41	8.52	2.56	1.19	0.66	6.93
Kshape	40.83	31.97	14.58	9.74	1.47	-1.79	2.16
Kvideo	38.20	8.59	4.11	3.11	1.39	0.59	1.68
Kwml	92.60	42.63	20.48	13.98	3.64	2.22	9.64
Scheduler	70.98	20.53	16.83	9.13	10.80	0.27	9.96
Average:	55.98	18.21	10.28	6.48	2.77	0.41	4.20



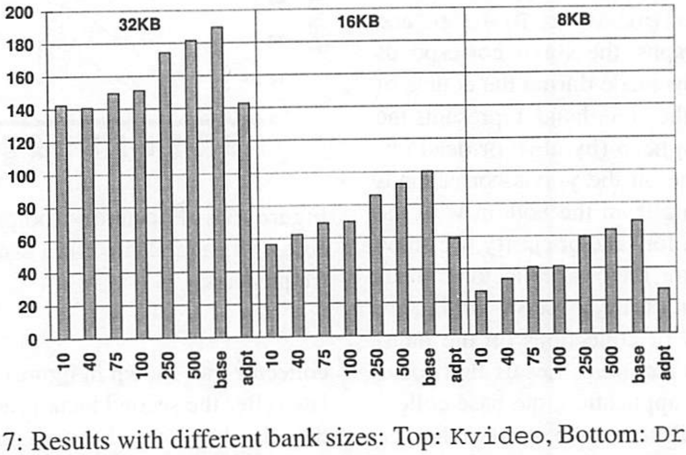


Figure 7: Results with different bank sizes: Top: Kvideo, Bottom: Dragon.

Table 2: The number of GC activations for each benchmark.

Application	10	40	75	100	250	500	base	adpt
Calculator	104	24	12	9	3	1	0	3
Crypto	595	147	78	58	23	11	3	26
Dragon	84	15	8	6	2	1	1	8
MathFP	825	205	109	81	32	16	2	21
ManyBalls	214	48	25	19	7	3	0	10
Kshape	2280	593	338	266	136	93	50	100
Kvideo	102	23	12	9	3	1	0	4
Kwml	4936	1099	571	425	167	83	19	285
Scheduler	973	240	127	95	37	18	2	28
Average:	1063.7	266	142.2	107.5	45.6	25.2	8.6	53.8

4.4 Performance Behavior

It is also important to evaluate the performance impact of our adaptive garbage collection strategy. Table 3 gives, for each version, the percentage increase in execution cycles with respect to the base collector. We see that the average (across all benchmarks) execution time increase due to our adaptive strategy is only 4.20%, which is less than the increases in execution times when k-allocation strategies with $k = 10, 40, 75,$ and 100 are employed. Considering the fact that our strategy is competitive with these allocators as far as the energy behavior is concerned, the performance overhead of the adaptive scheme is acceptable. As an example, in Dragon, our approach is competitive with the 10-allocation collector as far as the energy consumption is concerned. However, the 10-allocation collector increases the execution time of this application by 14.53%, a much larger figure than 0.68%, the corresponding increase in the adaptive collector.

4.5 Bank Size Variations

Recall that so far all experiments have been performed using a heap size of 128KB and a bank size of 16KB. In this subsection, we change the bank size (by keeping the heap size fixed at 128KB) and measure the sensitivity of our results to the bank size variation. Figure 7 gives the normalized heap energy consumptions for two of our benchmarks: Kvideo and Dragon. Our experiments with other benchmarks showed similar trends; so, they are not reported here in detail. The results given in Figure 7 are values normalized with respect to the heap energy consumption of the base allocator with 16KB bank size and 128KB heap size. We could not run Dragon with 4KB bank size due to the large inter-bank objects allocated by this benchmark (currently, our implementation cannot allocate objects larger than bank size). We observe two trends from this graph. First, the effectiveness of our adaptive collection strategy as well as that of k-allocation collectors increase with the reduced bank size. This is because a smaller bank size gives more opportunity to GC to turn off heap memory regions in a finer-grain manner. Consequently, more heap memory space can be turned off. Second, our adaptive strategy continues to follow the best k-allocation collector even when different bank sizes are used; that is, it can be used with different bank sizes.

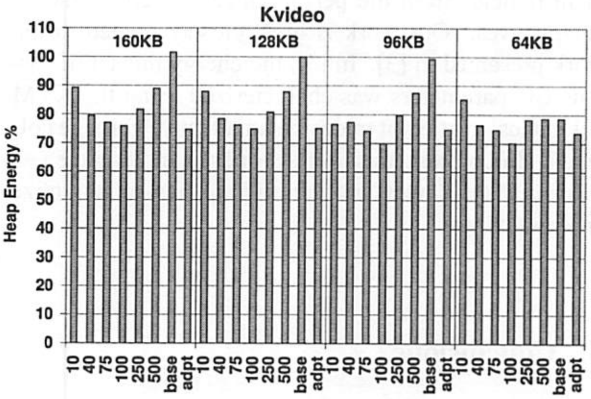


Figure 8: Results with different heap sizes (Kvideo).

4.6 Heap Size Variations

In this subsection, we fix the bank size at 16KB (our default value) and report experimental results obtained when different heap sizes are used. We focus on Kvideo only; but, the results observed with other benchmarks are similar. The heap sizes used in this set of experiments are 64KB, 96KB, 128KB (our default), and 160KB. We observe from the results shown in Figure 8 that our adaptive strategy continues to follow the best k-allocation collector and that its performance is not very sensitive to the heap size variation.

5 Related Work

Most efforts concerning the garbage collector and Java virtual machine have not focused on optimizing the energy consumption. However, there have been recent efforts at characterizing the energy behavior of Java applications. Flinn et al. [7] quantify the energy consumption of a pocket computer when running Java virtual machine. In [15], the energy behavior of a high-performance Java virtual machine is characterized. Diwan et al. [6] analyzed four different memory management policies from the performance as well as energy perspectives. Our work is most closely related to the work presented in [3]. In [3], the energy impact of various GC parameters was characterized using the KVM but makes no attempt to design an adaptive garbage collector. In contrast, this work proposes an adaptive GC strategy that tailors its invocation frequency to maximize energy savings.

6 Conclusions

Unnecessary leakage energy can be expended in maintaining the state of garbage objects until they are recognized to be unreachable using a collector. This wasted leakage energy is proportional to the duration between when an object becomes garbage and the time when it is recognized to become garbage. While invoking the GC more frequently can reduce this wasted leakage that needs to be balanced with the energy cost of invoking the GC itself. In this paper, we show the design of an adaptive GC strategy that tailors its invocation frequency to account for these tradeoffs based on the object allocation and garbage creation frequencies. We implemented this adaptive GC within a commercial JVM and showed that it is effective in reducing the overall system energy.

References

[1] L. Benini and G. De Micheli. System-level power optimization: techniques and tools. *ACM Transactions on Design Automation of Electronic Systems,* 5(2), pp.115-192, April 2000.

[2] A. Chandrakasan, W. J. Bowhill, and F. Fox. *Design of High-Performance Microprocessor Circuits.* IEEE Press, 2001.

[3] G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and M. Wolczko. Tuning garbage collection in an embedded Java environment. In Proc. *the 8th International Symposium on High-Performance Computer Architecture,* Cambridge, MA, February 2-6, 2002.

[4] CLDC and the K Virtual Machine (KVM). http://java.sun.com/products/cldc/.

[5] B. Cmelik and D. Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. In Proc. *ACM Sigmetrics Conference on the Measurement and Modeling of Computer Systems,* pp. 128-137, May 1994.

[6] A. Diwan, H. Li, D. Grunwald and K. Farkas. Energy consumption and garbage collection in low powered computing. http://www.cs.colorado.edu/~diwan. University of Colorado-Boulder.

[7] J. Flinn, G. Back, J. Anderson, K. Farkas, and D. Grunwald. Quantifying the energy consumption of a pocket computer and a Java virtual machine. In Proc. *International Conference on Measurement and Modeling of Computer Systems,* June 2000.

[8] W-M. W. Hwu. Embedded microprocessor comparison. http://www.crhc.uiuc.edu/IMPACT/ece412/public_html/Notes/412_lec1/ppframe.htm.

[9] R. Jones and R. D. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley and Sons. 1999.

[10] S. Kaxiras, Z. Hu, M. Martonosi. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. In Proc. *The 28th International Symposium on Computer Architecture,* June 2001.

[11] M. Kamble and K. Ghose. Analytical energy dissipation models for low power caches. In Proc. *International Symposium on Low Power Electronics and Design,* page 143, August 1997.

[12] M. D. Powell, S-H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar. Gated-Vdd: a circuit technique to reduce leakage in deep-submicron cache memories. In Proc. *the ACM/IEEE International Symposium on Low Power Electronics and Design,* August 2000.

[13] R. Riggs, A. Taivalsaari and M. VandenBrink. *Programming Wireless Devices with the Java 2 Platform.* Addison Wesley, 2001.

[14] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. Y. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using SimplePower. In Proc. *the International Symposium on Computer Architecture,* Vancouver, British Columbia, June 2000.

[15] N. Vijaykrishnan, M. Kandemir, S. Tomar, S. Kim, A. Sivasubramaniam and M. J. Irwin. Energy Characterization of Java Applications from a Memory Perspective. In Proc. *USENIX Java Virtual Machine Research and Technology Symposium,* April 2001.

Concurrent Remembered Set Refinement in Generational Garbage Collection

David Detlefs
Ross Knippel
Sun Microsystems[1]
david.detlefs@sun.com
ross.knippel@sun.com

William D. Clinger
Northeastern University
will@ccs.neu.edu

Matthias Jacob
Department of Computer Science
Princeton University
35 Olden Street
Princeton, NJ 08544
mjacob@cs.princeton.edu

Abstract

Generational garbage collection divides a heap up into two or more generations, and usually collects a *youngest* generation most frequently. Collection of the youngest generation requires identification of pointers into that generation from older generations; a data structure that supports such identification is called a *remembered set.* Various remembered set mechanisms have been proposed; these generally require mutator code to execute a *write barrier* when modifying pointer fields. Remembered set data structures can vary in their *precision*: an imprecise structure requires the garbage collector to do more work to find old-to-young pointers. Generally there is a tradeoff between remembered set precision and barrier cost: a more precise remembered set requires a more elaborate barrier. Many current systems tend to favor more efficient barriers in this tradeoff, as shown by the widespread popularity of relatively imprecise *card marking* techniques. This imprecision becomes increasingly costly as the ratio between old- and young-generation sizes grows. We propose a technique that maintains more precise remembered sets that scale with old-generation size, using a barrier whose cost is not significantly greater than card marking.

1 Introduction

Generational garbage collection [24] is a widespread and popular technique [30, 21, 32, 6, 3]. Generational collection usually both decreases average GC pause times (since most collections target just the youngest generation) and also increases GC efficiency (by concentrating collection work on the youngest generation, whose objects often die young). In collecting the youngest generation, objects in older generations are considered as roots; young objects reachable from old objects are considered live. Therefore, young objects reachable from older generations must be identified in order to have a correct collection. Further, many systems use some form of relocating garbage collection. For example, compaction enables fast linear allocation in a contiguous free space. This imposes a further requirement, that *all* pointers to young objects from other generations be identified, since those pointers must be updated.

Data structures that support iteration over old-to-young pointers (or pointers that cross other, more general boundaries) are often called *remembered sets.* Since mutator updates of pointer fields in objects may create new pointers that must be remembered, generational systems usually have an associated *write barrier* that is executed along with such updates to maintain the remembered set.

Remembered set implementations differ in their *precision:* how precisely they describe the locations of the pointer fields in old-generation objects that must be scanned. A highly precise remembered set leads to faster young-generation collection. A remembered set may be imprecise in two ways:

[1] Solaris and Java are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the United States and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

first, entries may only approximately describe the locations of cross-generational pointers, and second, some entries may not actually denote cross-generational pointers. Such imprecision leads to extra work during collection.

Unfortunately, there is a tradeoff between remembered set precision and the cost of the write barrier code executed by the mutator threads: a more precise remembered set generally requires a more elaborate, and therefore more expensive, write barrier. At one extreme, the "null" remembered set implementation would be extremely imprecise (scan the entire old generation to find cross-generational pointers), but impose no overhead on the mutator, since no barrier is necessary. At the other extreme, every pointer update could execute code to either insert or remove, as necessary, the updated location from a hash table representing the remembered set. Such a scheme would be very precise, but also very expensive.

Many remembered set/write barrier combinations have been proposed [33, 3]; these populate many points on this tradeoff curve. *Card marking,* in particular, has become a popular technique. In this technique, the heap is partitioned into equal-sized cards, and a *card table* array is allocated, with an entry for each card of the heap. Card table entries are initially *clean*; the mutator write barrier marks the card containing the updated field (or the head of the object containing the field, in a variant) *dirty.* The collector must scan the card table to find dirty entries, and then scan the corresponding dirty cards to find the cross-generational pointers, if any, created by the writes. This technique has a quite inexpensive write barrier (as few as 2 extra instructions per pointer update) and small memory overhead (a typical configuration has a one-byte card table entry for every 512-byte card). However, card marking is not particularly precise: the collector must scan the entire card table to find marked cards; cards may be marked by "false positive" pointer updates that create only intra-old-generation pointers; and even for dirty cards the collector may scan an entire card to find just one cross-generational pointer.[2] Many applications are requiring increasingly large heaps, while requiring pause times to remain small. Since the cost of card marking increases as a function of old-generation size and mutator pointer update rate, card marking may not scale to these larger heaps.

One positive trend is that applications with large heaps are increasingly multithreaded and are run on multiprocessors. However, tuning an application (and run-time system, and operating-system kernel) to scale with additional processors is often difficult. In this paper we propose that an available underutilized processor can be put to productive use by concurrent *refinement* of the remembered set to increase its precision. We investigate several strategies for remembered set representation, several alternative write barriers (two of them novel), and different concurrent processing methods. By using concurrent refinement, we are able to keep remembered set scanning in young-generation collection more nearly dependent only on the number of old-to-young pointers at the time of collection, and relatively independent of the size of the old generation or the application's pointer mutation rate. For one customer's application, concurrent refinement of the remembered set reduces both the average pause time and the total cost of garbage collection.

The rest of the paper is organized as follows. Section 2 discusses related work. In section 3 we describe the remembered sets we used. Section 4 describes the write barriers we considered. Section 5 describes the concurrent processing that uses the output of those barriers to refine the remembered set. Section 6 describes our experimental results. We close the paper with conclusions and future work.

2 Related work

In this section we describe related previous work. First we consider explorations of remembered set implementations and write barrier code sequences, then the application of concurrency to garbage collection.

2.1 Remembered sets and write barriers

Many remembered set representations have been proposed in the literature, with associated write barriers. Chapter 7.5 of Jones and Lins' garbage collection reference [21] contains an excellent

[2] Some card marking variants increase precision by increasing write barrier cost, for example, by filtering out old-to-old pointers in the write barrier.

overview.[3] The *sequential store buffer* scheme of Hosking, Moss, and Stefanović [19] is similar to our log-based barriers in that it separates data structures updated by mutator barriers from remembered set representations. However, the details of the mutator barrier code are different, and, while they processed logs incrementally as they overflowed, they did not process logs concurrently. The summary table mechanism used in our card-table-based remembered set implementation is similar to the the hybrid card marking scheme of Hosking and Hudson [20].

Sobalvarro [28] described a form of 2-level card table, but it required hardware support, and he did not address concurrent processing.

Recently, Fitzgerald and Tarditi [15] compared a number of different barrier and remembered set implementations including card marking, sequential store buffers, and a 2-level card table. Their conclusion was that, for their benchmarks, choice of write barrier did not greatly influence the performance of their system. They did not consider offloading work to a concurrent thread, as is done in this paper.

2.2 Concurrency

Concurrent garbage collection is not new. Steele [29] had an early algorithm. Dijkstra, Lamport, *et al.* introduced *on-the-fly* collection [10], a form of concurrent mark-and-sweep. This was extended by Kung and Song [22]. More recently, these ideas have been revisited for ML [11, 12] and for the Java™ programming language [14, 13]. Baker invented an incremental copying collector [5], which was implemented in hardware on Lisp machines [25]. Ellis, Li, and Appel implemented this idea on stock hardware with a virtual-memory-based barrier, and added true concurrency [23]. However, most of these have little relevance to the present work beyond the fact they involve garbage collection and concurrency.

Two other collection approaches are similar to the current work in that they use log-based write barriers whose output is processed by a concurrent thread devoted to GC. The first of these is *concurrent reference counting.* In this approach, the write barrier logs the address of the modified field, and its value before the modification. DeTreville [9] described such a collector for Modula-2+ (with a backup mark-sweep collector to detect garbage cycles). More recently, Bacon *et al.* [4] described another such system (with a concurrent local cycle-detection algorithm.) Logging is especially useful when this style of collection is applied to multi-threaded systems, since write barriers can write only to thread-local logs, and all modifications to object reference counts are done by a concurrent thread.

The other log-based collector we will mention is the *concurrent replicating collection* technique of O'Toole and Nettles [26]. In this approach all mutator updates (including those to non-pointer fields) are logged. A collector thread performs a concurrent copying collection. During collection, the mutator observes only from-space pointers. The GC thread ensures that logged updates are applied to to-space versions of already-copied objects, with pointers translated appropriately, and that updates that modify the pointer graph are handled correctly. This has little in common with the present paper beyond the use of logs and concurrency.

Another family of concurrent collectors with some relevance to the current work starts with the "mostly-parallel" collector of Boehm, Demers and Shenker [7]. Variations on this theme have been explored by Printezis and Detlefs [27] and by Heil and Smith [17]. The *process M* presented by Boehm *et al.* (termed *concurrent precleaning* in Printezis and Detlefs) could be considered a kind of concurrent remembered set refinement: in this application, the remembered set records pointers that have been modified during a concurrent marking phase, and whose referents therefore may not be marked. The concurrent process attempts to ensure that a necessary "stop-world" phase to complete the marking is short, much as the concurrent work in the current paper tries to make "stop-world" young-generation collections shorter.

While previous efforts bear some relation to the current ideas, none have explicitly had concurrent refinement of remembered set precision as a goal.

[3] Note, though, that we are departing somewhat from their terminology in this paper, and using *remembered set* to refer to the general class of data structures for recording cross-generational pointers, rather than a specific such data structure.

3 Remembered sets

We consider two remembered set organizations in this paper. The first is the default remembered set representation of the system in which we perform our measurements. This is a card table, augmented with a *summary table.* As described so far, a card table entry is either *clean,* indicating the absence of cross-generational pointers, or *dirty,* indicating their possible presence. When a young-generation collection scans a dirty card and finds a cross-generational pointer, and the collection does not *promote* the referent of that pointer out of the young generation, it leaves the card dirty to ensure that this pointer is also scanned in the next collection. The summary table makes this more efficient: the summary table can represent the positions of up to a maximum of k pointers within the card. If scanning of a dirty card finds k or fewer (but not zero) cross-generational pointers, then the card table entry is set to a new value *summarized* and the number and positions of those pointers are recorded in the summary table. Dirty cards containing more than k cards remain dirty; we refer to these as *overflow* cards.

The value k is a compile-time constant; how large must it be for summarization to be effective? Figure 1 contains histograms classifying cards by how many old-to-young pointers they contain at the time of collection, for each of the benchmarks described in section 6.2, summing over all collections. This is a "busy" figure; we don't intend for the reader to note fine details. But we will note that the great majority of cards are "clean," that is, contain no cross-generational pointers. Cards containing more than 16 cross-generational pointers are quite rare (note that the y-axis is logarithmic). Treating cards with more than 16 cross generational pointers as overflow cards has negligible cost.

One more technique can be used to decrease space overhead. As described so far, each card has a corresponding summary table able to hold k pointer offsets; in the implementation, each offset occupies a byte. In all but two of the benchmarks, no individual histogram bucket above the one for 2 pointers contains more than 100 cards (summed over all the collections that occur in the benchmark.) The other two benchmarks have quite large heaps, so the relative sizes of the histogram buckets are still small. Thus we could set k to 2, and use an encoding scheme in which a 2-byte summary table entry is identifiable as either a sequence of pointer offsets, or else an index into a separate *mid-size table,* which is able to represent 16 offsets. Few such entries will be necessary, and using $k = 3$ would ensure that more than 2 million would be addressable. We have not yet implemented this variant.

Note that the use of a card table as a remembered set does not require the use of a write barrier that updates the card table directly; our log-based barriers, for example, will use this representation also.

Our second remembered set organization is really just an enhancement of the first: it is a two-level card table. Each entry in the smaller, *coarse-grained* table corresponds to some number of entries in the larger, *fine-grained* table. (In our implementation, this ratio is always of the form 2^N, for some N.) A coarse-grained entry is clean only if all the corresponding fine entries are clean.

Clearly, a 2-level table scales better with large heaps. If a young-generation collection is required to scan an entire fine-grained table to find non-clean cards, it may spend a significant amount of time just skipping clean cards. Using the coarse-grained table speeds up scanning of large clean regions by a factor of 2^N.

One of the write barriers investigated below dirties both the coarse-grained and fine-grained cards corresponding to the updated location. A novel feature of our two-level table organization is the observation that by sacrificing a relatively small amount of address space, we can arrange to have a common *card table base* value for both tables. This speeds up the barrier code; see section 4.2 for details.

4 Write barriers

In this section we describe each write barrier that we have implemented by showing the barrier code that is executed following an assignment of the form `x.f = y` where `x` is a reference to an object and `y` is an expression of reference type. We will show the barrier code as SPARC® assembly language, using `%rx` to stand for a general register that contains `x`. We will use `%reg1`, `%reg2`, *etc.*, to stand for scratch registers, and the constant `foffset` to stand for the offset of field `x.f` from the address of `x` itself.

The write barriers that we have implemented fall

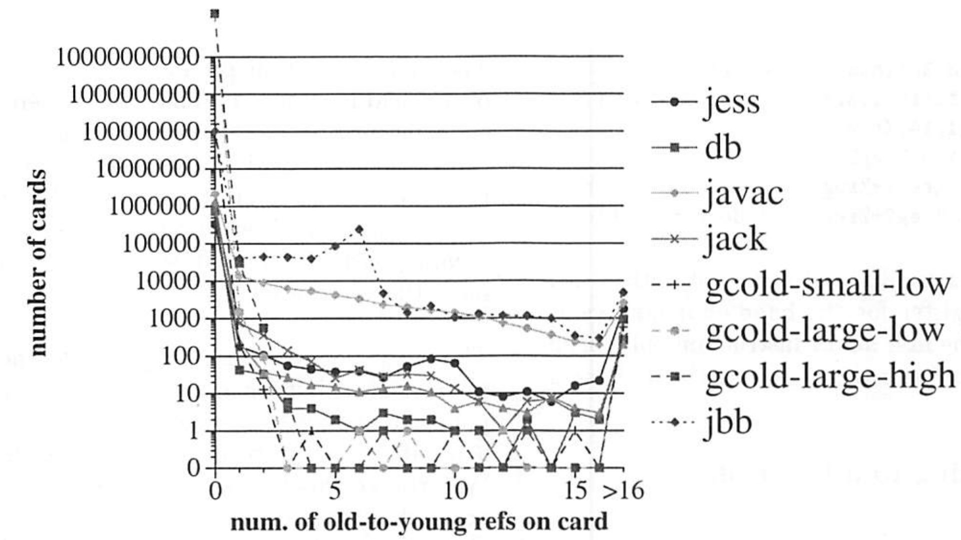


Figure 1: Histogram of number of old-to-young pointers on cards

into three categories:

1. barriers that directly update a 1-level card table
2. barriers that directly update a 2-level card table
3. barriers that adjoin an entry to a log buffer

4.1 Updating a 1-level Card Table

For our experiments, each byte of the 1-level card table represents $512 = 2^9$ bytes of the heap. Our straightforward *card-table* write barrier is

```
sethi   %reg3,%hi(base_address1) ! see below
add     %rx,foffset,%reg1    ! %reg1 = &x.f
srl     %reg1,9,%reg2
stb     %g0,[%reg2+%reg3]    ! mark card dirty
```

In the SPARC assembly code shown here, the `%g0` register always holds the constant zero; this value is used to represent a dirty card precisely because this register is available. This barrier code assumes that both the heap and card table are aligned on a 512-byte boundary, and that `base_address1` is related to the lowest heap address `H` and the lowest address of the card table `CT1` by

```
base_address1 = (CT1 - (H >> 9))
```

The barrier code also assumes that `CT1` has been aligned so that the low-order bits of `base_address1` are zero.

This barrier code sequence is similar to the two-instruction sequence of Hölzle [18].[4] We actually used a variant of this barrier in which, in any method that might execute a write barrier, a register is dedicated to holding the `base_address1` value, and the `sethi` instruction that initializes this register is executed once on method entry, not for every write barrier. The dedicated register is a SPARC local register, and is therefore preserved across calls by the the register window mechanism.

4.2 Updating a 2-level Card Table

For our 2-level card table we combined the 1-level card table with a coarse-grained card table in which each byte represents $16384 = 2^{14}$ bytes, or 32 cards in the fine-grained table. We aligned the address `CT0` of the coarse-grained table with respect to the fine-grained table so that

```
base_address1 = (CT1 - (H >> 9))
              = (CT0 - (H >> 14))
```

which allows us to use a single base register for both card tables. Our straightforward write barrier for the 2-level card table, which we will identify as *card-table2*, is

[4] Except that it marks the card containing the precise location of the modified field rather than the "imprecise" location of the object head, which is used in Hölzle's barrier. Hölzle's optimization saves an instruction and, more importantly a register; it is more important on register-poor architectures such as x86 than on a register-rich RISC architecture such as the SPARC. Using precise marking simplifies our GC code.

```
sethi   %reg3,%hi(base_address1)
add     %rx,foffset,%reg1     ! %reg1 = &x.f
srl     %reg1,14,%reg1
srl     %reg1,9,%reg2
stb     %g0,[%reg1+%reg3]     ! do coarse table
stb     %g0,[%reg2+%reg3]     ! do fine table
```

As before, we actually used a variant with a dedicated local register for the base_address1 value, initialized by the first sethi instruction only on entry to a method, instead of within the write barrier.

4.3 Adjoining to a Log Buffer

The design space becomes much larger when pointer writes are merely logged so that the remembered set can be updated later or by a concurrent log-processing thread. In this section we describe only two of the possible designs.

Both of the write barriers that we describe adjoin an entry to a thread-local write log buffer, which is essentially an array of heap locations that have been the left hand side of a pointer assignment. A global register %next is dedicated to point to the next entry in the log buffer. The main problem with this kind of barrier is that the log buffers can overflow, and explicit tests for overflow are expensive. One straightforward solution used in the past [19] is to terminate the log with a write-protected page and to detect overflow via a SIGSEGV exception. However, for overflow frequencies corresponding to reasonably-sized log buffers, we found that Unix signal handling is too expensive. Therefore, we investigated two barriers that handle log buffer overflow in other ways.

The first we call the *misalignment-utrap* barrier. The UTRAP mechanism of the Solaris^{TM} operating system, like UNIX signals, is a mechanism for handling hardware exceptions. It essentially handles only only misaligned accesses, but is about one hundred times as fast as the UNIX signal-handling mechanism for this exception. We therefore designed a barrier that performs a misaligned store (on a non-word boundary) when the log buffer overflows, and use the UTRAP mechanism to handle the misalignment exception. Here is the barrier:

```
add  %rx,foffset,%reg1  ! %reg1 = &x.f
srl  %next,n-1,%reg2    ! %reg2 = %next>>(n-1)
st   %reg1,[%next-4]
and  %reg2,6,%reg2      ! %reg2 = 4 or 6
add  %next,%reg2,%next  ! %next = %next + %reg2
```

The first instruction produces the precise address of the field modified. (An alternative version would elide the first instruction and log the object head rather than the field address, performing less mutator work at the cost of more work for concurrent refinement. This observation also applies to the *self-pointing* barrier described below. We did not implement this alternative.)

The %next register normally points to the next entry plus four, but when the log buffer is full %next points to the next entry plus six, which will cause a misalignment trap during the store instruction. We arrange this by using 2^n-byte log buffers that are each aligned on a 2^{n+1}-byte boundary but not aligned on a 2^{n+2}-byte boundary. The shift-right-logical and "and" instructions therefore generate the value 4 in %reg2, but generate a 6 when the buffer is full.

Our *self-pointing* barrier takes a quite different approach towards minimizing the cost of log buffer overflow: it attempts to avoid overflow altogether. Each mutator thread has an associated set of log buffers, linked in a sequence. Buffer entries are initially initialized with pointers to their own locations, except for the last entry of a buffer, which is initialized with the address of the first entry of the next buffer of the sequence (or NULL for the last entry of the last buffer). As with the misalignment-utrap barrier, a global %next register contains the address of the next log buffer entry to be written. The barrier stores through %next, then updates %next with the value read from the next entry:

```
add     %rx,foffset,%reg1 ! %reg1 = &x.f
st      %reg1,[%next]
ld      %next,[%next+4]
```

When NULL is read on one barrier and stored through on the next, a SIGSEGV handler adds a new log buffer to the thread's sequence. However, a concurrent refinement thread (see section 5) will continually be processing completed log buffers at the head of the thread's sequence. When it completes the head log buffer, it unlinks it from the thread's sequence and relinks it at the end. It also resets the final entry of the old last buffer to contain the address of the first entry of the new one, rather than NULL. If this happens in a timely manner, the mutator thread will never observe %next to contain the NULL value, and will never overflow a buffer. Adding log buffers when this does occur increases the buffer space devoted to the thread, decreasing the likelihood of future occurrences.

5 Concurrent refinement

For each (compatible) combination of one of the remembered set representations described in section 3 and one of the write barriers described in section 4, we implement a concurrent thread that processes the information produced by the barrier in order to produce a more precise remembered set. First we consider aspects of concurrent refinement common to all combinations, then we consider points specific to refinement with card-table and log-based barriers, respectively.

5.1 Common considerations for all barriers

At an abstract level of description, all of the concurrent processing functions sleep for some interval, then traverse the remembered set data structure, attempting to refine it (by eliminating false positives and/or making pointer location data more precise). When a collection occurs, the "abstract" remembered set must be considered to include any write barrier data structures, such as log buffers, not yet processed by the refinement thread. The collector starts by completing any such outstanding processing. Various heuristics may be used to control the sleep interval between concurrent refinement intervals. The goal of such heuristics is to simultaneously minimize both the outstanding work necessary to bring the "concrete" remembered set up-to-date at the beginning of collection, and also the CPU time used by the refinement thread. Such heuristics will generally be based on previous program history.

Clearly, a heuristic that attempts to aggressively "throttle back" concurrent refinement because little mutator activity has been observed recently is vulnerable to sudden increases in pointer mutation rate. Some of the barriers can decrease this vulnerability somewhat. The barriers can be classified according to whether or not they produce *overflow* events; in particular, we say that the misalignment-utrap barrier produces an overflow event whenever it fills a buffer and starts to use a new buffer.[5] The occurrence of overflow events can be used to trigger activity by the concurrent refinement thread: instead of sleeping for a fixed amount of time, the thread waits on an operating-system condition variable, using a timeout value to cause it to be resumed when the condition variable is signaled or the timeout expires, whichever comes first.

5.2 Barrier/remembered set compatibility

Which combinations are compatible? Each of the card table barriers requires the corresponding (1- or 2-level) remembered set representation, since they write directly to that representation. The log-based barriers, on the other hand, can be used with either remembered set representation, and, indeed, would be compatible with many others, since the barrier does not write directly to that representation.

5.3 Refinement with card-table barriers

Consider first concurrent refinement with the single-level card-table write barrier. The mutator will set some card table entries to *dirty*. The concurrent refinement thread traverses the card table, searching for dirty entries. When one is found, the refinement thread sets the entry to a new value, *refining*. It then scans the card for cross-generational pointers, computing a new value for the card: *clean*, *summarized* with some number of pointer offsets, or *overflow* (which we now distinguish from *dirty* to prevent repeated consideration of unmodified overflow cards in successive traversals). The thread then attempts to write the new value into the card table. To do so, it reads the current value, verifies that it is still *refining*, and uses a compare-and-swap (CAS) instruction to atomically change to the new value. A concurrent mutator operation may set the entry back to *dirty;* if so, the refinement is invalid, since it may have missed a pointer update, and is abandoned. We currently leave the card *dirty* and proceed to the next dirty card; we could also repeat the refinement process. We expect such contention for dirty cards to be sufficiently rare to make this choice irrelevant.

Concurrent refinement with a two-level card table barrier is very similar. The refinement thread searches the coarse-grained table for dirty entries.

[5] The self-pointing barrier can also be considered to produces a overflow event when it allocates a new buffer. But the whole point of the self-pointing barrier is to avoid such allocations, since they are triggered by dereferencing a null pointer, which invokes a relatively expensive signal handler. So these cannot be counted on as a reliable indicator of mutator activity.

When one is found it sets the coarse-grained entry to *refining* and searches the corresponding fine-grained entries. For each of those that are dirty, it goes through the process above. If all the fine-grained entries are or become *clean*, and the coarse-grained entry is still *refining*, then the coarse-grained entry is atomically reset to *clean*, otherwise it is reset (not atomically) to *dirty*.

There is a subtle concurrency issue involving the order in which the tables are updated by the barrier. The whole point of using a coarse-grained table is so that during a GC we can traverse the smaller coarse-grained table, and skip all the fine-grained cards corresponding to a clean coarse-grained card. In our system, garbage collections happen only at discrete *gc points*, which never occur during write barriers, so barriers (and their associated pointer updates) are atomic with respect to collection [2]. Thus a collection will never observe a partially completed 2-level barrier. Barriers and updates are not, however, atomic with respect to the actions of the concurrent refinement thread. It turns out that the barrier must update the fine-grained table before the coarse-grained table. With this order, the concurrent refinement thread may skip a clean coarse-grained card, where the mutator has just dirtied one of the corresponding fine-grained cards and is about to dirty the coarse-grained card. But, while this fails to achieve the maximum possible benefit of concurrent processing, it is still perfectly correct: the coarse-grained card will (correctly) be dirtied before the next GC. (And this situation is probably quite rare.) If the barrier is performed in the other order, an unfortunate scenario can result. Consider a clean coarse-grained card, all of whose covered fine-grained cards are also clean. Suppose a pointer update to a field in the last of these fine-grained cards does not create an old-to-young pointer, but nevertheless dirties the relevant fine-grained and coarse-grained cards. A second pointer update to a field in the first covered fine-grained card *does* create an old-to-young pointer, and the write barrier (redundantly) dirties the coarse-grained card. Now, before the second write barrier completes, the concurrent refinement thread observes the dirty coarse-grained card, scans all the covered fine-grained cards, and finds only the last dirty. It scans that card, finds that it does not contain an old-to-young pointer, and therefore resets both the fine-grained and coarse-grained cards to clean. At this point, the partially-completed second barrier completes, dirtying the first fine-grained card. At this point, the invariant is violated, and the barrier is complete, so a GC could occur and observe this violation. Therefore, the 2-level barrier must dirty the fine-grained card first.

5.4 Refinement with log-based barriers

We now consider concurrent refinement with log-based barriers. Each thread has an associated *thread log set,* which contains log buffers associated with that thread. A thread log set is created and initialized as part of thread creation, before the thread can execute any write barriers. We also maintain a global set of all the thread log sets; initialization of a thread log set includes insertion of the new set into that global set. The refinement thread can iterate over this global set of thread log sets. Concurrent insertion of new thread log sets may cause those to be skipped, but the initial log processing at the start of garbage collection uses sufficient synchronization to ensure that no non-empty logs are skipped. Thread logs may contain unprocessed entries when the thread completes. Therefore, the thread log set is not deleted when the corresponding thread is dead; rather, it is marked as "dead," implying that no more entries will be written to its log buffers. When the refinement thread completes processing of a dead log set, it deletes it from the global set, and frees its storage.

In the misalignment-utrap barrier, there is also a global list of completed log buffers. The refinement thread processes all completed buffers, and may also traverse the thread log sets, processing partially completed buffers.

As described in section 4, log entries are addresses within the heap. These can be distinguished from non-entries in both logging schemes, so the refinement thread can tell when it has read all the currently-valid entries in a log buffer. A valid entry may or may not represent an old-to-young pointer. For each entry, the refinement thread first considers the address of the field. Write barriers are executed for all objects, so some of the logged addresses may be in young-generation objects; these can be ignored.

The refinement thread next reads the current value of the field. It is important to note that in all our barriers, the barrier must be executed *after* the write it covers, or else the refinement thread might observe the log entry or modified card, but read the field value *before* the write, and take an improper action.[6] There is no guarantee that the value read by the refinement thread when it processes a log entry will be the value written by the mutator thread that added that entry, both because a thread may update a location several times, and because a location may be updated by several distinct threads. However, since we require the refinement thread to process all log entries for a location, and the last write to a location happens before the last entry for that location is logged, we are guaranteed that the last entry processed for a given location (either by the refinement thread, or by the collector during its initial log processing) will observe the *final* write to a given pointer field.

If the pointer value is not a pointer into the young generation, then the log entry is ignored. This is a design choice; we have chosen to have the remembered set be monotonically non-decreasing between collections. We could have alternatively attempted to detect when pointer updates decrease the size of the remembered set, but we judged that this would have created significantly more work for the refinement thread for a small benefit.

6 Results

In this section we present measurements of the effectiveness of concurrent refinement. Section 6.1 describes the system in which we performed our preliminary experiments, section 6.2 describes the benchmarks, and sections 6.3 and 6.4 report the results.

Following those experiments, we transferred this technology to a product group. Section 6.5 describes their implementation of concurrent refinement and summarizes its impact on one customer's application.

6.1 Experimental system

We implemented concurrent refinement by modifying the Sun Microsystems Laboratories Virtual Machine for Research, henceforth *ResearchVM*, a high performance Java^{TM} virtual machine[7] developed by Sun Microsystems. This virtual machine has been previously known as the "Exact VM", and has been incorporated into products; for example, the Java^{TM} 2 SDK (1.2.1_07) Production Release, for the Solaris operating environment.

The ResearchVM features high-performance *exact* (i.e., non-*conservative* [8], also called *precise*) memory management [1]. The memory system is separated from the rest of the virtual machine by a well-defined *GC Interface* [31]. This interface allows different garbage collectors to be "plugged in" without requiring changes to the rest of the system. A variety of collectors implementing this interface have been built. In addition to the GC interface, a second layer, called the *generational framework*, facilitates the implementation of generational garbage collectors. These interfaces allowed us to parameterize the implementation over the various choices of remembered sets and barriers relatively easily.

The default configuration of this system uses a two-generation collector, with a semispace-based young generation, and an older generation that uses mark-sweep-compact collection.

Measurements were performed on an otherwise idle Sun Enterprise^{TM} E6500 server with 16 400 MHz UltraSPARC® II processors sharing 16 GB of memory.

6.2 Benchmarks

We measured the performance of this first implementation on several benchmarks.

The first benchmark is a synthetic one written by the authors, called **gcold**.[8] This application takes several command-line flags that control the workload it presents to the collector. In particular, we can create workloads that require a large old-generation, and also vary the pointer mutation rate. We use this application to demonstrate the existence of workloads for which concurrent refinement shows a significant advantage. This application has been found to be predictive of real application performance in the past. We vary two parameters: heap

[6] As discussed previously, other mechanisms ensure that the entire field-write/write-barrier combination is atomic with respect to GC.

[7] The term "Java virtual machine" means a virtual machine for the Java^{TM} platform.

[8] Earlier versions of this benchmark have been used in other studies of concurrent [27] and parallel [16] collection.

size and pointer mutation rate in the old generation. The *small* heap size is 30 MB live in a 45 MB heap, and *big* is 300 MB live in a 450 MB heap. The *low* pointer mutation rate is less than 1000 old-generation pointers updated per second, and *high* is approximately 300,000 old-generation pointers updated per second of mutator operation. (For comparison, **javac** updates about 70,000 old-generation pointers per second of mutator time. A multi-threaded program with behavior similar to **javac** could easily have an aggregate pointer mutation rate this large.) We ran the following **gcold** configurations: **gcold-S-L** (small/low), **gcold-B-L** (big/low), and **gcold-B-H** (big/high).

The next benchmark is called **jbb**; this is a SPEC benchmark aimed at measuring performance of Java virtual machines executing server applications on multiprocessors. It has large heaps and requires significant GC activity. Unlike the others, which measure the time necessary to accomplish some fixed task, this is a "throughput-oriented" benchmark, measuring how many iterations of a repetitive task can be executed in a fixed amount of time. We make the measurements commensurate by restricting ourselves to the first 500 garbage collections. (The different configurations measured do not differ in the timing of allocation and young-generation collection.) This was run with a 16 MB young generation and a 300 MB old generation.

The remaining four benchmarks are the members of the SPECjvm98 suite that spent more then 3% of their elapsed time performing collection in our system with its default configuration. These are

- **javac**: a compiler that translates Java programming language source code to Java class files, compiling a given set of files;
- **jess**: an "expert systems shell" written in the Java language, solving a set of logic problems;
- **mtrt**: a "multi-threaded ray tracer," in which two worker threads render an image; and
- **jack**: a parser generator.

These four run in heaps of at most 24 MB.

6.3 Measurements

Table 1 shows the performance of our benchmarks averaged over five runs for each of the various system configurations. The configurations are as follows:

- CT1: the default one-level card table barrier and remembered set.
- CT1+C: one-level card table barrier and remembered set with concurrent refinement.
- CT2: two-level card table barrier and remembered set, no concurrent refinement.
- CT2+C: two-level card table with and remembered set, with concurrent refinement.
- SP-CT1: self-pointing logging barrier, concurrent refinement, one-level card table remembered set.
- SP-CT2: self-pointing logging barrier, concurrent refinement, two-level card table remembered set.
- MIS-CT1: misalignment-utrap logging barrier, concurrent refinement, one-level card table remembered set.
- MIS-CT2: misalignment-utrap logging barrier, concurrent refinement, two-level card table remembered set.

All configurations allow up to 16 summary-table entries per card.[9]

6.4 Discussion

We would first direct the reader's attention to the columns for young-generation collection time and time to find cross-generation pointers; these are the aspects of collection that we are trying to improve via concurrent refinement.

The **gcold-B-H** run shows that there exist applications (admittedly synthetic) for which the improvement can be dramatic. The **jbb** benchmark is somewhat less artificial, having been written to model a class of real programs; here we see as much as a 5%

[9] This is to allow more direct comparisons; the standard configuration of the ResearchVM allows only 2 such entries.

Benchmark	system configuration	total (sec)	mutator (sec)	old-gen gc (sec)	young-gen gc (sec)	cross-gen ptrs (sec)
gcold-S-L	CT1	101.5	56.3	33.5	11.6	0.6
	CT1+C	101.2	56.7	32.9	11.6	0.4
	SP-CT1	103.6	58.1	33.8	11.7	0.4
	MIS-CT1	102.8	58.8	33.0	10.9	0.4
	CT2	101.0	56.6	32.9	11.6	0.6
	CT2+C	101.4	56.6	33.7	11.1	0.5
	SP-CT2	103.2	58.3	32.9	12.0	0.3
	MIS-CT2	104.5	59.7	33.9	11.0	0.3
gcold-B-L	CT1	113.0	60.4	25.8	26.8	1.6
	CT1+C	115.9	64.1	25.0	26.8	1.3
	SP-CT1	117.3	64.2	26.2	26.8	1.2
	MIS-CT1	117.5	66.3	25.0	26.2	1.2
	CT2	112.5	60.8	25.1	26.7	1.2
	CT2+C	112.8	61.0	26.0	25.8	0.8
	SP-CT2	117.0	65.5	24.9	26.6	0.8
	MIS-CT2	120.2	68.3	26.2	25.8	0.8
gcold-B-H	CT1	257.4	167.6	12.2	77.6	60.5
	CT1+C	215.8	184.3	11.7	19.9	2.5
	SP-CT1	199.0	168.6	12.5	17.9	0.8
	MIS-CT1	202.0	172.4	11.7	17.9	0.9
	CT2	260.8	171.1	11.7	78.0	60.7
	CT2+C	216.6	184.8	12.3	19.5	2.4
	SP-CT2	199.5	170.2	11.6	17.7	0.6
	MIS-CT2	204.3	174.0	12.5	17.8	0.7
jbb	CT1	311.4	189.4	22.3	99.8	23.4
	CT1+C	314.3	197.1	21.7	95.5	19.1
	SP-CT1	312.1	192.7	21.8	97.6	18.6
	MIS-CT1	329.5	212.2	22.2	95.1	18.7
	CT2	334.5	211.2	23.8	99.5	22.8
	CT2+C	329.2	211.8	22.8	94.6	18.2
	SP-CT2	313.6	195.9	22.3	95.4	17.7
	MIS-CT2	343.0	226.4	23.0	93.6	17.7
javac	CT1	35.6	28.1	3.5	4.0	1.4
	CT1+C	34.5	27.5	3.4	3.6	0.9
	SP-CT1	36.4	29.1	3.5	3.7	0.9
	MIS-CT1	37.4	30.3	3.5	3.6	0.9
	CT2	35.2	27.7	3.4	4.1	1.4
	CT2+C	35.0	27.9	3.5	3.6	0.9
	SP-CT2	36.1	28.9	3.5	3.7	0.9
	MIS-CT2	37.6	30.5	3.5	3.6	0.9
jess	CT1	17.0	16.3	0.0	0.7	0.1
	CT1+C	16.4	15.6	0.0	0.7	0.1
	SP-CT1	20.5	19.6	0.0	0.9	0.1
	MIS-CT1	20.8	20.0	0.0	0.7	0.1
	CT2	16.7	16.0	0.0	0.7	0.1
	CT2+C	17.0	16.2	0.0	0.7	0.1
	SP-CT2	21.1	20.2	0.0	0.9	0.1
	MIS-CT2	21.6	20.8	0.0	0.8	0.1
mtrt	CT1	9.9	9.1	0.2	0.6	0.0
	CT1+C	9.7	9.0	0.2	0.5	0.0
	SP-CT1	10.1	9.3	0.2	0.6	0.0
	MIS-CT1	9.8	9.0	0.2	0.5	0.0
	CT2	10.0	9.3	0.2	0.5	0.0
	CT2+C	9.5	8.8	0.2	0.5	0.0
	SP-CT2	10.1	9.3	0.2	0.6	0.0
	MIS-CT2	9.8	9.0	0.2	0.5	0.0
jack	CT1	22.5	21.1	0.1	1.3	0.0
	CT1+C	22.4	21.1	0.1	1.3	0.0
	SP-CT1	23.7	22.2	0.1	1.4	0.0
	MIS-CT1	23.4	22.0	0.1	1.3	0.0
	CT2	22.1	20.7	0.1	1.3	0.0
	CT2+C	22.3	20.9	0.1	1.3	0.0
	SP-CT2	23.3	21.8	0.1	1.4	0.0
	MIS-CT2	23.5	22.1	0.1	1.3	0.0

Table 1: Comparison of elapsed, mutator, and gc times for various system configurations.

decrease in young-generation collection times. The **javac** benchmark is based on a real program, and shows up to 10% improvements in young-generation collections. The other benchmarks have few cross-generational pointers, and show little if any benefit from concurrent refinement.

The six configurations that use concurrent refinement show little variation with respect to young-generation collection times.

Improvements due to concurrent refinement do not come without cost.

The default CT1 barrier executes three instructions, including one write, per pointer write (not counting the `sethi` per method entry). The more complicated barriers (CT2, SP, and MIS) increase mutator time significantly. The SP barrier performs more memory operations, and most of its increased mutator time is due to data cache misses. The CT2 and MIS execute more instructions and have more instruction cache misses. Larger caches would reduce these costs. A more general UTRAP mechanism (that supports segmentation violations as well as misalignment exceptions) would give us a logging barrier with the same instruction count and memory operations as the default CT1 barrier.

In addition, we find that concurrent card table refinement can increase mutator time even with the default card table barrier, as in the CT1+C configuration on the **gcold-B-H** and **jbb** benchmarks. Both of these benchmarks have a high rate of pointer mutation, and the concurrent refinement thread runs almost continuously. We suspect this leads to data cache contention. With the two-level card table (CT2+C), the refinement thread's duty cycle on the **jbb** benchmark drops from 91% to 72%, eliminating this effect. For concurrent refinement with the logging barriers, the duty cycle ranged from 11% (SP-CT1 on **db**) to 82% (MIS-CT2 on **jbb**).

We had expected more improvement from CT2 *vs.* CT1 for large heaps. We believe the lack of such a gain is due to CT1 using an optimized assembly-language routine to recognize 64-bit blocks of clean cards. This loop is not currently used in CT2; it could be, and should result in more speedups for very sparse card tables. In any case, the advantage of the two-level card table will become important only on very large heaps.

Old-generation collection time is not affected by concurrent refinement. The variation in old-generation collection time for the three **gcold** benchmarks is due to different heap sizes, numbers of collections, object lifetimes, and patterns of floating garbage.

6.5 Production system

We have been working with a telecommunications company that has a call-processing application written in the Java language. In its steady state, this program has several hundred megabytes of live data. When a mostly-concurrent mark/sweep collector is used to collect the old generation, pause times are dominated by the time required to collect the young generation [27]. In the terminology of this paper, this system used the CT1 barrier and remembered set. To reduce pause times still further, a product group added concurrent refinement, to get the equivalent of CT1+C, in a limited-release version of the ResearchVM.

Using the original CT1 barrier adds no mutator overhead, other than processor time taken by the concurrent refinement thread. The main problem with our experimental implementation had been that this processor time had been considerable. To reduce that overhead, the new version does not initiate concurrent refinement until there is barely enough time to complete one or two passes of concurrent refinement before the young generation is collected. For example, concurrent refinement might begin when 90% of the young generation has been allocated, but this threshold is adjusted dynamically.

When the telecommunications application is run on a machine with eight processors, the concurrent refinement thread runs less than 2% of the time, while reducing the average pause time by about 15%. All mutator threads are stopped while the young generation is collected, and those collections had accounted for about 15% of the total time, so concurrent refinement increases the mutator's utilization of the processors by about 2%:

$$.02 \approx .15 \cdot .15 - \frac{.02}{8}$$

In other words, concurrent refinement simultaneously reduces both average pause times and the total cost of garbage collection.

7 Conclusions

The desirable pause time characteristics of generational collection will not scale with ever-increasing heap sizes, at least using currently-popular remembered set techniques. Programs with very large heaps are likely to be run on machines with many processors. We have therefore suggested ways to use concurrency to retain short GC pauses, by refining the precision of remembered set representations so that cross-generational pointers can be found quickly.

We have presented two basic refinement techniques: "direct" refinement of one- and two-level card tables (which has interesting non-blocking concurrency control) and log-based refinement, in which mutator threads log updates and the refinement thread applies those, as appropriate, to a representation of the remembered set. In the latter case, we presented two novel write barrier code sequences that minimize the frequency and/or cost of log buffer overflow.

Concurrent refinement techniques allow generational collection to scale to future systems that will have extremely large heaps and high aggregate pointer mutation rates.

8 Acknowledgments

Several colleagues at Sun Microsystems have contributed to this general set of ideas. Bernd Matthiske and Ross Knippel originally proposed a version of the self-pointing barrier, using the UTRAP mechanism to trigger flushing of small thread-local log buffers to a large common one. They did not consider concurrent refinement. Dave Dice suggested the use of a self-pointing barrier with concurrent refinement to avoid overflow.

References

[1] O. Agesen and D. Detlefs. Finding references in Java™ stacks. In *Proceedings of the OOPSLA'97 Workshop on Garbage Collection and Memory Management*, Atlanta, GA, USA, October 1997.

[2] Ole Agesen. GC points in a threaded environment. Technical Report 98-70, Sun Microsystems Laboratories, 1998.

[3] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2):171–183, 1989.

[4] David Bacon, Clement Attanasio, Han Lee, V. T. Rajan, and Stephen Smith. Java without the coffee breaks: a nonintrusive multiprocessor garbage collector. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI-01)*, volume 36.5 of *ACM SIGPLAN Notices*, pages 92–103, N.Y., June 20–22 2001. ACM Press.

[5] H. G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[6] Henry G. Baker. "Infant mortality" and generational garbage collection. *SIGPLAN Notices*, 28(4):55–57, April 1993.

[7] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In Brent Hailpern, editor, *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 157–164, Toronto, ON, Canada, June 1991. ACM Press.

[8] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, September 1988.

[9] John DeTreville. Experiences with concurrent garbage collectors for Modula-2+. Technical Report 64, Digital Equipment Corporation Systems Research Center, 1990.

[10] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *CACM*, 21(11):966–975, November 1978.

[11] D. Doligez and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 113–123, New York, NY, 1993. ACM.

[12] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Conference Record of the Twenty-first Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 70–83, New York, NY, USA, January 1994. ACM Press.

[13] Tamar Domani, Elliot K. Kolodner, Ethan Lewis, Elliot E. Salant, Katherine Barabash, Itai Lahan, Erez Petrank, Igor Yanover, and Yossi Levanoni.

Implementing an on-the-fly garbage collector for Java. In Tony Hosking, editor, *Proceedings of the Second International Symposium on Memory Management*, Minneapolis, MN, October 2000. ACM Press.

[14] Tamar Domani, Elliot K. Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for Java. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 274–284, Vancouver, British Columbia, June 18–21, 2000.

[15] Robert Fitzgerald and David Tarditi. The case for profile-directed selection of garbage collectors. In Tony Hosking, editor, *Proceedings of the Second International Symposium on Memory Management*, Minneapolis, MN, October 2000. ACM Press.

[16] Christine H. Flood, David Detlefs, Nir Shavit, and Xiaolan Zhang. Parallel garbage collection for shared memory multiprocessors. In *Proceedings of the Java™ Virtual Machine Research and Technology Symposium*, Monterey, April 2001. USENIX.

[17] Timothy H. Heil and James E. Smith. Concurrent garbage collection using hardware-assisted profiling. In *Proceedings of the International Symposium on Memory Management*, Minneapolis, Minnesota, October 15–19, 2000.

[18] Urs Hölzle. A fast write barrier for generational garbage collectors. In Eliot Moss, Paul R. Wilson, and Benjamin Zorn, editors, *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems*, October 1993.

[19] A. L. Hosking, J. E. B. Moss, and D. Stefanovic. A comparative performance evaluation of write barrier implementations. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 92–109, Vancouver, Canada, October 1992.

[20] Antony L. Hosking and Richard L. Hudson. Remembered sets can also play cards. In *ACM OOPSLA'93 Workshop on Memory Management and Garbage Collection*, Washington, DC, October 1993.

[21] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley & Sons, Ltd, 1996.

[22] H. T. Kung and S. Song. An efficient parallel garbage collector and its correctness proof. Technical report, Carnegie Mellon University, September 1977.

[23] John R. Ellis; Kai Li; and Andrew W. Appel. Real-time concurrent collection on stock multiprocessors. Technical Report 25, Digital Equipment Corporation Systems Research Center, February 1988.

[24] Henry Lieberman and Carl E. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983. Also report TM–184, Laboratory for Computer Science, MIT, Cambridge, MA, July 1980 and AI Lab Memo 569, 1981.

[25] David A. Moon. Garbage collection in a large Lisp system. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 235–246. ACM, August 1984.

[26] James O'Toole and Scott Nettles. Concurrent replicating garbage collection. In *Conference on Lisp and Functional programming.* ACM Press, June 1994.

[27] Tony Printezis and David Detlefs. A generational mostly-concurrent garbage collector. In *Proceedings of the International Symposium on Memory Management*, Minneapolis, Minnesota, October 15–19, 2000.

[28] Patrick G. Sobalvarro. A lifetime-based garbage collector for Lisp systems on general-purpose computers. B.S. thesis, Massachusetts Institute of Technology EECS Department, Cambridge, Massachusetts, 1988.

[29] Guy L. Steele Jr. Multiprocessing compactifying garbage collection. *CACM*, 18(9):495–508, September 1975.

[30] D. M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984.

[31] D. White and A. Garthwaite. The GC interface in the EVM. Technical Report TR-98-67, Sun Microsystems Laboratories, 1999.

[32] P. R. Wilson. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 1–42, St. Malo, France, September 1992. Springer-Verlag.

[33] Paul R. Wilson and Thomas G. Moher. A "card-marking" scheme for controlling intergenerational references in generation-based garbage collection on stock hardware. *ACM SIGPLAN Notices*, 24(5):87–92, May 1989.

To Collect or Not To Collect?
Machine Learning for Memory Management

Eva Andreasson
BEA/Appeal Virtual Machines
Folkungagatan 122
S-102 65 Stockholm
eva.andreasson@appeal.se, d97-eva@d.kth.se

Frank Hoffmann
Centre for Autonomous Systems
Royal Institute of Technology
S-100 44 Stockholm
hoffmann@nada.kth.se

Olof Lindholm
BEA/Appeal Virtual Machines
Folkungagatan 122
S-102 65 Stockholm
olof.lindholm@bea.com

ABSTRACT

This article investigates how machine learning methods might enhance current garbage collection techniques in that they contribute to more adaptive solutions. Machine learning is concerned with programs that improve with experience. Machine learning techniques have been successfully applied to a number of real world problems, such as data mining, game playing, medical diagnosis, speech recognition and automated control. Reinforcement learning provides an approach in which an agent interacts with the environment and learns by trial and error rather than from direct training examples. In other words, the learning task is specified by rewards and penalties that indirectly tell the agent what it is supposed to do instead of telling it how to accomplish the task. In this article we outline a framework for applying reinforcement learning to optimize the performance of conventional garbage collectors.

In this project we have researched an adaptive decision process that makes decisions regarding which garbage collector technique should be invoked and how it should be applied. The decision is based on information about the memory allocation behavior of currently running applications. The system learns through trial and error to take the optimal actions in an initially unknown environment.

1 Introduction

JRockit™, the Java™ Virtual Machine (JVM) constructed by Appeal Virtual Machines and now owned by BEA and named Weblogic JRockit, was designed recognizing that all applications are different and have different needs. Thus, a garbage collection technique and a garbage collection strategy that works well for one particular application may perform poorly for another. To achieve good performance over a broad spectrum of different applications, various garbage collection techniques with different characteristics have been implemented. However, any garbage collection technique requires a strategy that allows it to adapt its behavior to the current context of operation. Over the past few years, the need for better and more adaptive strategies has become apparent.

Imagine that a JVM is running a program X. For this program, it might be best to garbage collect according to a rule Y. Whenever Y becomes true, the JVM garbage collects. However, this might not be the optimal strategy for another program X'. For X', rule Y' might be the best choice. Combining rule Y and Y' does not have to be complicated, but consider writing a combined rule that works really well for hundreds of programs? How does the JVM implementer know that a rule that works really well for many programs doesn't perform badly on others? Providing startup parameters for controlling the rule heuristics is a good start but it cannot adapt over time to a dynamic environment that has different needs at different points of time.

The idea is to let a learning decision process decide which garbage collector technique to use and how to use it, instead of static rules making these decisions during run time. The learning decision process selects among different kinds of state of the art garbage collection techniques in JRockit™, the one that is best suitable for the current application and platform.

The objective for this investigation is to find out if machine learning is able to contribute to improved performance of a commercial product. Theoretically machine learning could contribute to more adaptive solutions, but is such an approach feasible in practice?

This paper is concerned with the question whether and, if so, how a learning decision process can be used for a more dynamic garbage collection in a modern JVM, such as JRockit.

1.1 Paper Overview

Section 2 relates the paper to previous work and in Section 3 we present the problem specification. Section 4 provides a survey of the reinforcement learning method that has been used. Section 5 presents possible situations of a system that uses a garbage collector in which a learning decision process might perform better than a regular garbage collector. Section 6 handles the design of the prototype and is followed by a presentation of experimental results, discussion of future developments and conclusions in Section 7, 8 and 9.

2 Related work

To our current best knowledge we are not aware of any other attempt to utilize reinforcement learning in a JVM. Therefore, we are not able to provide references to similar approaches for that particular problem. Many papers on garbage collection techniques include some sort of heuristics on when the technique should be applied, but they are usually quite simple. These methods are usually straightforward and based on general rules that do not take the specific characteristics of the application into account.

Brecht et al. [7] provide an analysis on when garbage collection should be invoked and when the heap should be expanded in the context of a Boehm-Demers-Weiser (BDW) collector. However, they do not introduce any adaptive learning but instead investigate the characteristic properties of different heuristics.

3 Problem Specification

The problem to solve is: how to design an automatic and learning decision process for more dynamic garbage collection in a modern JVM.

Unlike some other garbage collection techniques, such as parallel garbage collection and stop-and-copy, concurrent garbage collection starts to garbage collect before the memory heap is full. A full heap would cause all application threads to stop, which would not be necessary if the concurrent garbage collector had started in time, since a concurrent garbage collector allows running applications to run concurrently with some phases of the garbage collection. For further reading about garbage collection, see references [2, 6, 8, 9, 13, 14].

An important issue, when it comes to concurrent garbage collection in a JVM, is to decide when to garbage collect. Concurrent garbage collection must not start too late, or else the running program may run out of memory. Neither must it be invoked too frequently, since this causes more garbage collections than necessary and thereby disturbs the execution of the running program. The key idea in our approach is to find the optimal trade-off between time and memory resources by letting a learning decision process decide when to garbage collect [2, 6, 8, 9, 13, 14].

4 Reinforcement Learning

Reinforcement learning methods solve a class of problems known as Markov Decision Processes (MDP). If it is possible to formulate the problem at hand as an MDP, reinforcement learning provides a suitable approach to its solution [3, 4, 5].

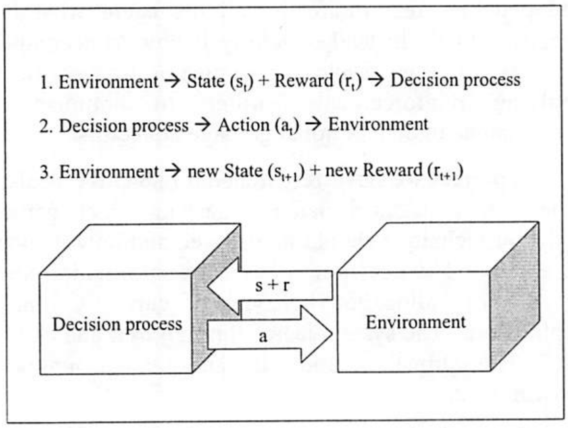

1. Environment → State (s_t) + Reward (r_t) → Decision process
2. Decision process → Action (a_t) → Environment
3. Environment → new State (s_{t+1}) + new Reward (r_{t+1})
s + r
a
Decision process
Environment

Figure 1 *The figure shows model of a reinforcement learning system. First the decision process observes the current state and reward then the decision process performs an action that effects the environment. Finally the environment returns the new state and the obtained reward.*

Figure 1 depicts the interaction between an agent and its environment in a typical reinforcement learning setting. The agent perceives the current state of the environment by means of the state signal s_t upon which it responds with a control action a_t.

More formally, a policy is a mapping from states to actions π: SxA $\rightarrow$ [0, 1], in which $\pi(s, a)$ denotes the probability with which the agent chooses action *a* in state *s*. As a result of the action taken by the agent in the previous state, the environment transitions to a new state s_{t+1}. Depending on the new state and the previous action the environment might pay a reward to the agent. The scalar reward signal indicates how well the agent is doing with respect to the task at hand. However, reward for desirable actions might be delayed, leaving the agent with the temporal credit assignment problem of figuring out which actions lead to desirable states of high rewards. The objective for the agent is to choose those actions that maximize the sum of future discounted rewards:

$$R = r_t + \gamma\, r_{t+1} + \gamma^2\, r_{t+2} \ldots.$$

The discount factor $\gamma \in [0,1]$ favors immediate rewards over equally large payoffs to be obtained in the future, similar to the notion of an interest rate in economics [1, 3, 5].

Notice, that usually the agent knows neither the state transition nor the reward function, neither do these functions need to be deterministic. In the general case the system behavior is determined by the transition probabilities $P(s_{t+1} \mid s_t, a_t)$ for ending up in state s_{t+1} if the agent takes action a_t in state s_t and the reward probabilities $P(r \mid s_t, a_t)$ for obtaining reward *r* for the state action pair s_t, a_t.

A state signal that succeeds in retaining all relevant information about the environment is said to have the Markov property. In other words, in an MDP the probability of the next state of the environment only depends on the current state and the action chosen by the agent, and does not depend on the previous history of the system [1, 3, 5].

A reinforcement learning task that satisfies the Markov property is an MDP. More formally: if *t* indicates the time step, *s* is the state of the environment, *a* is an action taken by the agent and *r* is a reward, then the environment and the task have the Markov property if and only if [5]:

$\Pr\{s_{t+1} = s', r_{t+1} = r \mid s_t, a_t\}$ is equal to:

$$\Pr\{s_{t+1} = s', r_{t+1} = r \mid s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0\}$$

If it is possible to define a way of representing states such that all relevant information for making a decision is retained in the current state, the garbage collection problem becomes an MDP. Therefore, a prerequisite for being able to use reinforcement learning methods successfully is to find a way to represent states in a correct manner [1, 3, 5].

In theory it is required that the agent has complete information about the state of the environment in order to be able to guarantee asymptotic convergence to the optimal solution. However, often fast learning is much more important than a guarantee of eventually optimal performance. In practice, many reinforcement learning schemes are still able to achieve a good behavior in a reasonable amount of time even if the Markov property is violated [10].

Whereas dynamic programming requires a model of the environment for computing the optimal actions, reinforcement learning methods are model free and the agent obtains knowledge about its environment through interaction. The agent explores the environment in a trial and error fashion, observing the rewards obtained of taking various actions in different states. Based on this information the agent updates its beliefs about the environment and refines its policy that decides what action to take next [4, 5].

4.1 Temporal-Difference Learning

There are mainly four different approaches to solve Markov decision processes: Monte Carlo, temporal-difference, actor-critic and R-learning. For further discussion about these methods, see references [5, 6, 12, 15].

What distinguishes temporal-difference learning methods from the other methods is that they update their beliefs at each time step. In application environments where the memory allocation rate varies a lot over time, it is important to observe the amount of available memory at each time step. Hence temporal-difference learning seems to be well suited for solving the garbage collecting problem [3, 5, 11, 15].

Temporal-difference learning is based on a value function, referred to as *the Q-value function*, which calculates the value of taking a certain action in a certain state. The algorithm performs an action, observes the new state and the achieved reward at each time step. Based on the observations, the algorithm updates its beliefs – the policy – and thereby theoretically improves its behavior at each time step [3, 5, 11, 15].

There are mainly two different approaches when it comes to temporal-difference methods: Q-learning and SARSA (State, Action, Reward, new State, new Action). This project has investigated the SARSA approach, since it is an on-policy method. On-policy means updating the policy that is being followed, i.e. the policy improves while being used. Further issues regarding how to use this method are discussed below.

4.2 Exploring vs. Exploiting

In reinforcement learning problems the agent is confronted with a trade-off between exploration and exploitation. On the one hand it should maximize its reward by always choosing the action $a = \max_a Q(s, a')$ that has the highest Q-value in the current state *s*. However, it is also important to explore other actions in order to learn more about the environment. Each time the agent takes an action it faces two possible alternatives. One is to execute the action that according to the current beliefs has the highest Q-value. The other possibility is to explore a non-optimal action with a lower expected Q-value of higher uncertainty. Due to the probabilistic nature of the environment, an uncertain action of lower expected Q-value might ultimately turn out to be superior to the current best-known action. Obviously there is a risk, that the taking of the sub-optimal action diminishes the overall reward. However, it still contributes to the knowledge about the environment, and therefore allows the learning program to take better actions with more certainty in the future [4, 5, 11, 12].

There are three different types of exploration strategies for choosing actions, the greedy algorithm, the ε-greedy algorithm and the soft-max algorithm. The greedy algorithm is not of interest to use, since the garbage collection problem requires exploration. Both the other two algorithms are well suited for the garbage collection problem. However, the ε-greedy algorithm was the choice we made.

The ε-greedy algorithm chooses the calculated, best action most of the times, but with a small probability ε a random action is selected instead. The probability of choosing a random action is decreased over time and hence satisfies both needs for exploration and exploitation [1, 5].

4.3 Generalization

Another common problem is environments that have continuous, and consequently infinitely many states. In this case it is not possible to store state-action values (Q-values) in a simple look-up table. A look-up table representation is only feasible when states and actions are discrete and few. Function approximation and generalization are solutions to this problem [3, 12].

Generalization is a way of handling continuous values of state features. As it is the case of the garbage collection problem, generalization of the state is needed. Alternative approaches, other than generalization, to approximate the Q-value function are regression methods and neural networks [4, 6]. However, the approach used during this project was generalization.

There are mainly four approaches for generalizing states and actions: coarse coding, tile coding, radial basis functions and Kanerva coding. For further reading about these methods see references [3, 5, 6].

Coarse coding is a generalization method using a binary vector, where each index of the vector represents a feature of the state, either present (1) or absent (0). Tile coding is a form of coarse coding where the state features are grouped together in partitions of the state space. These partitions are called *tilings*, and each element of a partition is called a *tile*. The more tilings you have, the more states will be affected of the reward achieved and share the knowledge obtained from an action performed. On the other hand, the system will get exponentially more complex depending on how many tilings are used [3, 5].

Tile coding is particularly well suited for use on sequential digital computers and for efficient online learning and is therefore used in this project [5].

5 State Features and Actions of the General Garbage Collection Problem

In the sections below some state features, actions and underlying reward features, possible to apply in a memory management system, are presented. Discussions of how they may be represented are also provided.

5.1 Possible State Features

A problem in defining state features and rewards for a Markov decision process, is the fact that the evolution of the state to a large extent is governed by the running application as it determines which objects on the heap are no longer referenced and how much new memory is allocated. The garbage collector can only partially influence the amount of available memory in that it reduces fragmentation of the heap and frees the memory occupied by dead objects. Therefore, it is often difficult to decide whether to blame the garbage collecting strategy or the application itself for exceeding the available memory resources.

In the following sections we present some suggestions of possible state features. Some state features might be difficult to calculate accurately at run time. For example, if the free memory were distributed across several lock-free caches, the number of free bytes would be hard to measure, or would at least take prohibitively long time to measure correctly. We therefore have to assume that approximations of these parameters are still accurate enough to achieve a reasonably good behavior.

A fragmentation factor that indicates what fraction of the heap is fragmented is of interest. *Fragments* are chunks of free memory that are too small (<2kB) to belong to the free-list, from which new memory is allocated. As the heap becomes highly fragmented, garbage collection should be performed more frequently. This is desirable as it might reduce fragmentation by collecting dead objects adjacent to fragments. As a result, larger blocks of free memory may appear that can be reused for future memory allocation. In other words garbage collection should be performed when the heap contains a large number of non-referenced, small blocks of free memory.

It is important to keep track of how much memory is available in the heap. Based on this information the reinforcement learning system is able to decide at which percentage of allocated memory it is most rewarding to perform a certain action, for instance to garbage collect.

If the rate at which the running program allocates memory can be determined, it would be possible to estimate at what point in time the application will run out of memory, and hence when to start garbage collection at the latest.

If it is possible to estimate how much processor time is actually spent on executing instructions of the running program, this factor could be used as a state feature. However, when using a concurrent garbage collector it is very difficult to measure the exact time spent on garbage collection versus the time used by the running application. Hence, this measurement will either be impossible to obtain or the information is highly inaccurate.

The average size of newly allocated objects might provide valuable information about the application running that can be utilized by the garbage collector. Another feature of the same category is the average age of newly allocated objects, if measurable. The amount of newly allocated objects is another possible feature.

5.2 State Representation

Each observable system parameter, described in the previous section, constitutes a feature of the current state. Tile coding, see Section 4.3, is used to map the continuous feature values to discrete states. Each tiling partitions the domain of a continuous feature into tiles, where each tile corresponds to an interval of the continuous feature.

The entire state is represented by a string of bits, with one bit per tile. If the continuous state value falls within the interval that constitutes the tile, the corresponding bit is set to 'one', otherwise it is set to 'zero':

- The tile contains the current state feature value → 1
- The tile does not contain the current state feature value → 0

For example, a particular state is represented by a vector s = [1, 1, 0, …, 1, 0, 1], where each bit denotes the absence or presence of the state feature value in the corresponding tile.

5.3 Possible Rewards

To evaluate the current performance of the system, quantifiable values of the goals of the garbage collector are desired. The objectives of a garbage collector (see references [6, 9, 13 14]) concern maximization of the end-to-end performance and minimization of long interruptions of the running application, caused by garbage collection. These goals provide the basis for defining the appropriate scalar rewards and penalties.

A necessity when deciding the reward function is to decide what are good and bad states or events. In a garbage-collecting environment there are a lot of situations that are neither bad nor good per se but might ultimately lead to a bad (or good) situation. This dynamic aspect adds another level of complexity to the environment. It is in the nature of the problem that garbage collection always intrudes on the process time of the running program and always constitutes extra costs. Therefore, no positive rewards are given but all reinforcement signals are penalties for consuming computational resources for garbage collection or even worse: running of out of memory. The objective of the learning process is to minimize the discounted accumulated penalties incurred over time.

A fundamental rule for imposing penalty is to punish all activities that consume processing time from the running program. For instance a punishment is imposed every time the system performs a garbage collection. An alternative is to impose a penalty proportional to the fraction of time spent on garbage collection compared to the total run time of the program.

Another penalty criterion is to punish the system when the average pause time exceeds an upper limit that is considered still tolerable by the user. It is also important to assure that the number of pauses does not exceed the maximum allowed number of pauses. If the average pause time is high and the number of pauses is low, the situation may be balanced by taking less time-consuming actions more frequently. If they are both high, a penalty might be in order.

When using a concurrent collector, a severe penalty must be imposed if the running program runs out of memory and as a result has to wait until a garbage collection is completed, since this is the worst possible situation to arise.

At first, it seems like a good idea to impose a penalty proportional to the amount of occupied memory. However, even if the memory is occupied up to 99 % this does not cause a problem, as long as the running application terminates without exceeding the available memory resources. In fact, this is the most desirable case, namely that the program terminates requiring no garbage collection but still never runs out of memory. Therefore, directly imposing penalties for the occupation of memory is not a good idea.

The ratio of freed memory after completed garbage collection compared to the ratio allocated memory in the heap prior to garbage collection provides another possible performance metric. This parameter gives an estimate of how much memory has been freed. If the amount is large there is nothing to worry about, as illustrated to the left in Figure 2. If the amount freed memory is low and the size of the free-list is low as well, problems may occur and hence the garbage collector should be penalized. The latter situation, illustrated to the right in Figure 2, might occur if a running program has a lot of long-living objects and runs for a long time, so that most of the heap will be occupied.

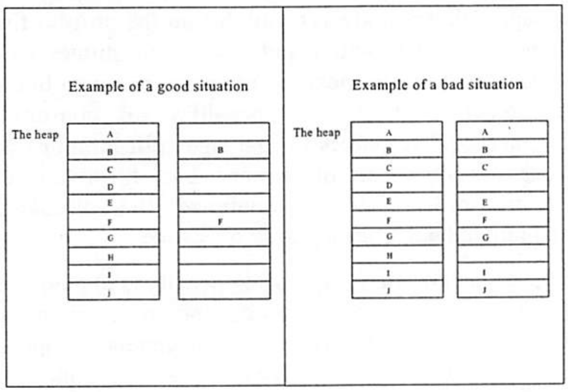


Figure 2 *A good situation with a high freeing rate is illustrated to the left. A worse situation is illustrated to the right, where there is little memory left in the heap although a garbage collection has just occurred. This last situation may cause problems.*

When using compacting garbage collectors, it is interesting to observe the success rate of allocated memory in the most fragmented area of the heap. The actual amount of new memory allocated in the fragmented area of the heap is compared to the theoretical limit of available memory in case of no fragmentation at all. An illustration of some possible situations is shown in Figure 3. It is desirable that 100 % of the newly allocated memory is allocated in the most fragmented area of the heap, in order to reduce fragmentation. A penalty is imposed that is inversely proportional to the ratio of actual allocated memory and its theoretical limit in the best possible case.

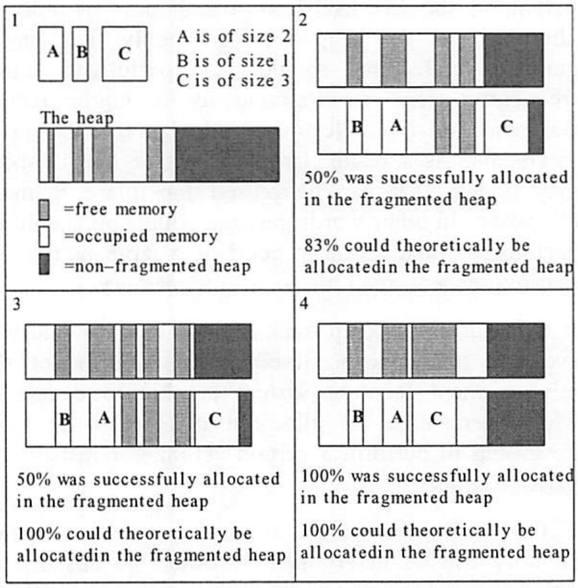

1
A
B
C
A is of size 2
B is of size 1
C is of size 3
The heap
=free memory
=occupied memory
=non–fragmented heap
2
B
A
C
50% was successfully allocated in the fragmented heap
83% could theoretically be allocatedin the fragmented heap
3
B
A
C
50% was successfully allocated in the fragmented heap
100% could theoretically be allocatedin the fragmented heap
4
B
A
C
100% was successfully allocated in the fragmented heap
100% could theoretically be allocatedin the fragmented heap

Figure 3 *To the upper right (2) half of the new allocated memory was successfully allocated in the fragmented heap. To the lower left (3) the same percentage was successfully allocated in the fragmented heap although space for all new allocated objects exists in the fragmented area. To the lower right (4) all new allocated objects were successfully allocated in the fragmented heap.*

If the memory relies on global structures that need a lock to be accessed, taking the lock ought to be punished. This might be the case for memory free-lists, caches etc.

The more time a compacting garbage collector spends on iterating over the free-list (for explanation see references [13, 14]) the more it should be penalized. A long garbage collection cycle is an indicator for a fragmented heap. High fragmentation in itself is not necessarily bad, but the iteration consumes time otherwise available to the running application, which is why such a situation should be punished.

When it comes to compacting garbage collectors a measurement of the effectiveness of a compaction provides a possible basis for assigning a reward or a penalty. If there was no need for compacting, the section in question must have been non-fragmented. Accordingly a situation like this should be assigned a reward.

There is one possible desirable configuration to which a reward, rather than a penalty, should be assigned, namely if a compacting collector frees large, connected chunks of memory. The opposite, if the garbage collector frees a small amount of memory and the running program is still allocating objects, could possibly be punished in a linear way, as some of the other reward situations described above.

5.4 Possible Actions

Whether to invoke garbage collection or not at a certain point of time is the most important decision for the garbage collecting strategy to take. Therefore, the set of possible actions taken by the prototype discussed in the later section is reduced to this binary decision.

When the free memory is not large enough and the garbage collection fails to free a sufficiently large amount of memory, a possible remedy is to increase the size of the heap. It is also of interest to be able to decrease the heap size, if a large area of the heap never becomes allocated. To decide whether to increase or decrease the heap size can constitute an action. If a change is needed a complementary decision is to decide the new size of the heap.

To save heap space or rather to use the available heap more effectively, a decision to compact the heap or not, could also be of interest. In addition the action could specify how much and which section of the heap to compact.

To handle synchronization between allocating threads of the running program, a technique of using lock-free Thread Local Areas (TLAs) is usually used. Each allocating thread is allowed to allocate memory within only one TLA at a time and vice versa there is only one thread permitted to allocate memory in a particular TLA. The garbage collection strategy could determine the size of each TLA and how to distribute the TLAs between the threads.

When allocating large objects often a Large Object Space (LOS) is used, especially in cases where generational garbage collectors are considered, in order to avoid moving large objects. Deciding the size of the LOS and how large an object has to be, to be considered a large object, are additional issues for the reinforcement learning decision process to consider.

To reduce garbage collection time, smaller free blocks might not be added to a free list during a sweep-phase. The memory block size is the minimum size of a free memory block for being added to the free list. Different applications may have different needs with respect to this parameter.

How many generations are optimal for a generational garbage collector? With the current implementation it is only possible to decide prior to starting the garbage collector if it operates with either one or two generations. It might be possible, even today, to reduce the number of generations from two to one, but not to increase them during run-time. When it comes to future generational garbage collectors it would be of interest to let the system vary the size of the different generations. If there is a promotion rate available, this is a factor that might be interesting for the system to vary as well.

If the garbage collector uses an incremental approach, deciding the size of the heap area that is collected at a time might be an interesting aspect to consider. The same applies to deciding whether to use the concurrent approach, in conjunction with the factors of how many garbage collection steps to perform at a time and how long a time the system should pre-clean (for explanation see references [14]).

6 The Prototype

The state features used in the prototype are the current amount of available memory s_1 and the change in available memory s_2, calculated as the difference between s_1 *at the previous time step* - s_1 *at the current time step*.

There is only one binary decision to make, namely whether to garbage collect or not. Hence, the action set contains only two actions {0, 1}, where 1 represents performing a garbage collection and 0 represents not performing a garbage collection.

The tile coding representation of the state in the prototype was chosen to be one 10x2-tiling in the case where only s_1 was used. In the case where both state features were used the tile coding representation was chosen to be one 10x7x2-tiling, one 10-tiling, one 7-tiling and one 10x7-tiling. A non-uniform tiling was chosen, in which the tile resolution is increased for states of low available memory, and a coarser resolution for states in which memory occupancy is still low. The tiles for feature s_1 correspond to the intervals [0, 4], [4, 8], [8, 10], [10, 12], [12, 14], [14, 16], [16, 18], [18, 20], [22, 26] and [30, 100]. The tiles for feature s_2 are at a resolution: [<0], [0-2], [3-4], [5-6], [7-8], [9-10] and [>10].

The reward function of the prototype imposes a penalty (-10) for performing a garbage collection. The penalty for running out of memory is set to -500. It is difficult to specify the quantitative trade-off between using time for garbage collection and running out of memory. In principle the later situation should be avoided at all costs, but a too large penalty in that case might bias the decision process towards too frequent garbage collection. Running out of memory is not desirable since a concurrent garbage collector is used. A concurrent garbage collector must stop all threads if the system runs out of memory, which is the major purpose of using a concurrent garbage collector in the first place.

The probability p that determines whether to pick the action with the highest Q-value or a random action for exploration evolves over time according to the formula:

$$p = p_0 * e^{-(t / C)}$$

where $p_0 = 0.5$ and $C = 5000$ in the prototype, which means that random actions are chosen with decreasing probability until approximately 25000 time steps elapsed. A time step *t* corresponds to about 50ms of real time between two decisions of the reinforcement learning system.

The learning rate α decreases over time according to the formula stated below:

$$\alpha = \alpha_0 * e^{-(t / D)}$$

where $\alpha_0 = 0.1$ and $D = 30000$ in the prototype. The discount factor γ is set to 0.9.

The test application used for evaluation is designed to demonstrate a very dynamic memory allocation behavior. The memory allocation rate of the test application alternates randomly between different *behavior cycles*. A behavior cycle consists of either 10000 iterations or 20000 iterations of either low or high memory allocation rate. The time performance of the RLS is measured during a behavior cycle as the number of milliseconds required to complete the cycle.

6.1 Interesting Comparative Measurements

The performance of the garbage collector in JRockit ought to be compared to the performance when using the reinforcement system for deciding when to garbage collect not only in terms of time performance but also in terms of the reward function. The reward function is based on the throughput and the latency of a garbage collector and the underlying features of the reward function are hence suitable for extracting comparable results of the two systems.

However, learning a proper garbage collection policy should take a reasonable amount of time, as otherwise the reinforcement learning system would be of little practical value. The first step of an evaluation of RLS is to verify that learning and adaptation actually occur at all, namely that the system improves its performance over time. The learning success is measured by the average reward per time step. Analyzing the time evolution of the Q-function provides additional insight into the learning progress.

7 Results

One of the main objectives of this project is the identification of suitable state features, underlying reward features and action features for the dynamic garbage-collection learning problem. An additional objective is the implementation of a simple prototype and the evaluation of its performance on a restricted set of benchmarks in order to investigate whether the proposed machine learning approach is feasible in practice.

This section compares the performance of a conventional JVM with a JVM using reinforcement learning for making the decision: when to garbage collect. The JVM using reinforcement learning is referred to as the RLS (Reinforcement Learning System) and the conventional JVM is JRockit.

Since JRockit is optimized for environments in which the allocation behavior changes slowly, environments where the allocation behavior changes more rapidly might cause a degraded performance of JRockit. In these environments it is of special interest to investigate if an adaptive system, such as an RLS, is able to perform equally well or even better than JRockit.

Figure 4 shows the results of using the RLS and JRockit for the test application described in Section 6. Due to the random distribution of behavior cycles a direct cycle-to-cycle comparison of these two different runs is not meaningful. Instead, the accumulated time performances, illustrated in Figure 4, are used for comparison. As may be seen in the lower chart, the RLS performs better than JRockit in this dynamic environment. This confirms the hypothesis of an RLS being able to outperform an ordinary JVM in a dynamic environment.

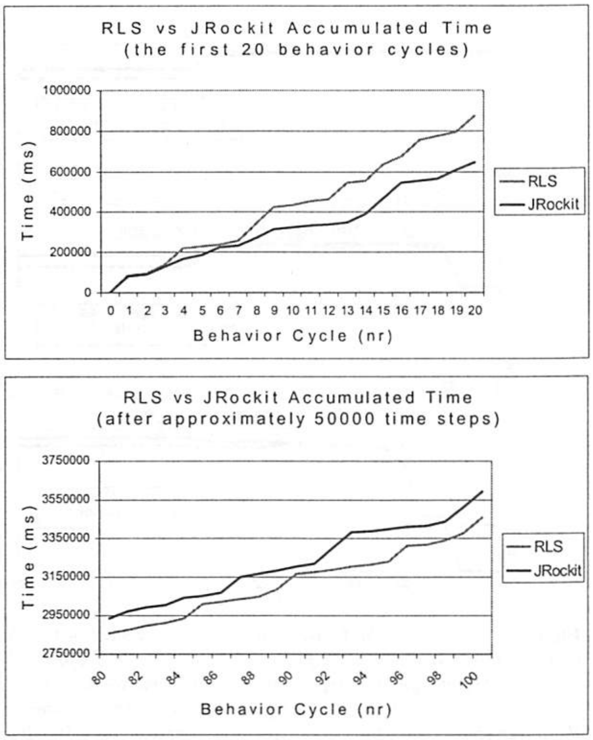


Figure 4 *The figure illustrates the accumulated time performance of the RLS and JRockit when running the application with behavior cycles of random duration and memory allocation rate. The upper chart shows the performances during the first 20 behavior cycles and the lower chart shows the performances during 20 behavior cycles after approximately 50000 time steps. Notice that lower values correspond to better performance.*

Figure 5 illustrates the accumulated penalty for the RLS compared to JRockit. In the beginning the RLS runs out of memory a few times, as shown in the graph labeled penalty RLS for running out of memory, but after about 15000 time steps it learns to avoid running out of memory. The lower chart shows the current average penalty of the RLS and JRockit. After about 20000 time steps the RLS has adapted its policy and achieves the same performance as JRockit. The results show that the RLS in principle is able to learn a policy that can compete with the performance of JRockit. The test session only takes about an hour, which is a reasonable learning time for offline learning (i.e. following one policy while updating another) of long running applications. Also, no attempt has been made to optimize the parameters of the RLS, such as exploration and learning rate, in order to minimize learning time within this project.

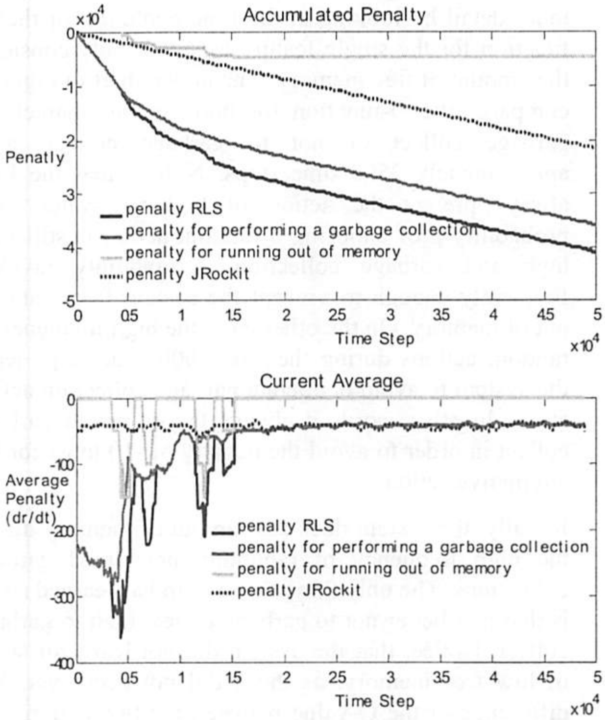


Figure 5 *The upper chart illustrates the accumulated penalty for the RLS compared to JRockit. The lower chart illustrates the average penalty as a function of time. For RLS the penalty due to garbage collection and due to running out of memory is shown separately.*

The accumulated penalty over a time period between time step 30000 and 50000 after RLS completed learning, has been calculated to -8400. The corresponding accumulated penalty for JRockit for the same period of time was calculated to -8550. This shows that the results of the RLS are comparable to the results of JRockit. The values verify the results presented above: that the RLS performs equally well or even slightly better than JRockit in an intentionally dynamic environment.

In the following we analyze the learning process in more detail by looking at the time evolution of the Q-function for the single feature case that only considers the amount of free memory. The upper chart in Figure 6 compares the Q-function for both actions, namely to garbage collect or not to garbage collect, after approximately 2500 time steps. Notice, that the RLS always prefers the action of higher Q-value. The probability p of choosing a random action is still very high and garbage collection is randomly invoked frequently enough to prevent the system from running out of memory. On the other hand the high frequency of random actions during the first 5000 time steps leads the system to avoid deliberate garbage collection action at all. In other words it always favors not to garbage collect in order to avoid the penalty of -10 units for the alternative action.

Initially, the system does not run out of memory due to the high frequency of randomly performed garbage collections. The only thing the system has learned so far is that it is better not to garbage collect than to garbage collect. Notice, that the system did not learn for states of low free memory, as those did not occur yet. The difference of the Q-value between the two actions is -10, which corresponds exactly to the penalty for performing a garbage collection. This makes sense insofar as the successor state after performing a garbage collection is similar to the state prior to garbage collection, namely a state for which the amount of memory available is still high.

The middle chart in Figure 6 shows the Q-function after approximately 10000 time steps. The probability of choosing a random action has now decreased to the extent, that the system actually runs out of memory. Once that happens the RLS incurs a large penalty, and thereby learns to deliberately take the alternative action, namely to garbage collect at states of low available memory.

The lower chart in Figure 6 illustrates the Q-function after approximately 50000 time steps. At this point the Q-values for the different states has already converged. Garbage collection is invoked once the amount of available memory becomes lower than approximately 12%. This policy is optimal considering the limited state information available to RLS, the particular test application and the specific reward function.

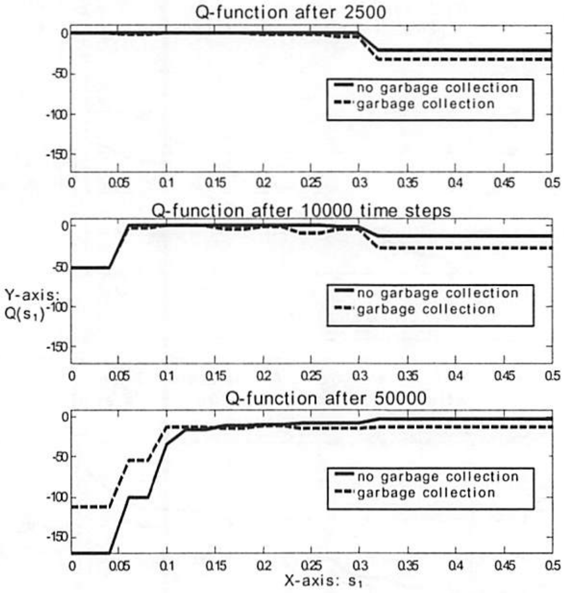


Figure 6 *The figure shows the development of the state-action value function, the Q-function, over time. The upper chart shows the Q-function after approximately 2500 time steps. The middle chart shows the Q-function after approximately 10000 time steps and the lower chart shows the Q-function after approximately 50000 time steps and is then constant.*

The performance comparison between the RLS and JRockit suggests further investigation of reinforcement learning for dynamic memory management. Regarding the fact that this first version of the prototype only considers a single state feature, it would be interesting to investigate the performance of an RLS that takes additional and possibly more complex state features into consideration. Additional state features might enable the RLS to take more informed decisions and thereby achieve even better performance.

In Figure 7 the accumulated time performance of the RLS using one (1F2T) and two state features (2F5T), and JRockit (JR) is compared. In the case of two state features, five (instead of only two) tilings were used in order to achieve better generalization across the higher dimensional state space. In order to illustrate the effect of five tilings, the time performance of an RLS using two state features but only two tilings (2F2T) is also shown in the charts of Figure 7. The upper chart illustrates the performance of the four systems in the initial stage at which the RLS is adapting its policy. The lower chart shows the performance after approximately 50000 decisions (time steps). The graphs show that the RLS using two state features and five tilings does not perform better than the RLS using only one state feature or JRockit. However, the system using five tilings is significantly better than the RLS using two state features and two tilings.

The main reason for the inferior behavior is probably that the new feature increases the number of states and that therefore converging to the correct Q-values and optimal policy requires more time. The decision boundary is more complex than in the case of only a single state feature. The number of states for which the RLS has to learn that it runs out of memory, if it does not perform a garbage collection, has increased and thereby also the complexity of the learning task.

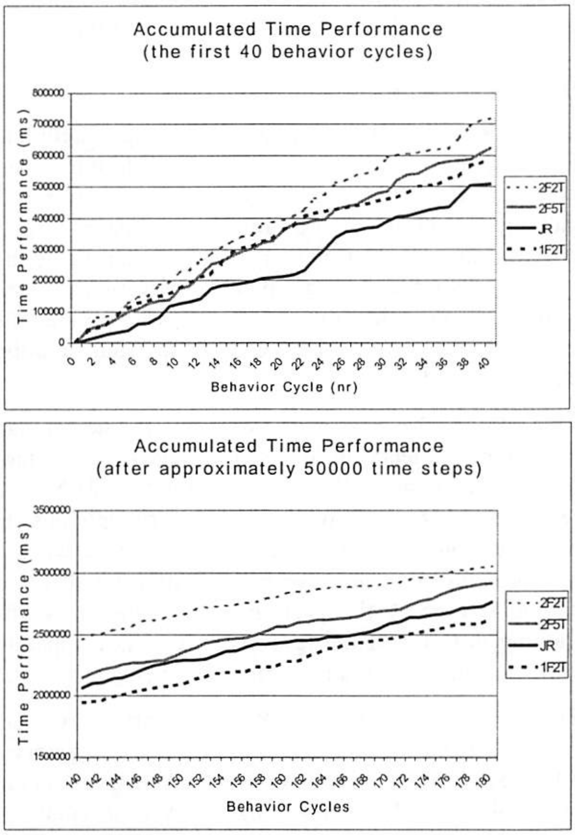


Figure 7 *The figure shows the accumulated time performance of JRockit compared to the RLS using one state feature and two RLS using two state features but different tilings.*

Another consequence of the increased number of states is that the system runs out of memory more often. To some extent Q-function approximation (i.e. tile coding, function approximation) provides a remedy to this problem. Further investigation regarding this aspect is needed, see the discussion in Section 8.

To provide some standard measurement results the best RLS, i.e. the RLS using only one state feature, is compared to the JRockit version used in previous test sessions due to SPECjbb2000 scores. In Figure 8 the results of a test session with full occupancy from the beginning are presented. As mentioned before, the RLS is learning until the 30000th time step (decision).

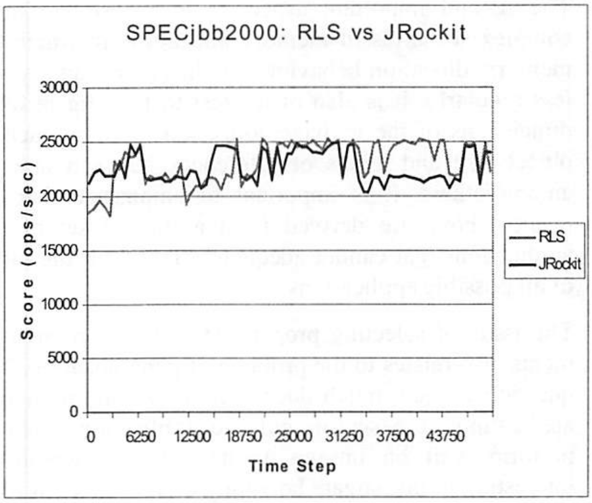


Figure 8 *The figure illustrates the performance of the RLS using one state feature compared to JRockit of a SPECjbb2000 session with full occupancy from the beginning.*

The average performance scores of both systems are presented in Table 1. As may be observed, the use of the RLS for the decision of when to garbage collect improves the average performance by 2%. That number already includes the learning period. If the learning period of the RLS is excluded (i.e. measured after approximately 30000 decisions), the average improvement when using the RLS is 6%.

Table 1 *The table illustrates the average performance results of the RLS using one state feature and JRockit, when running SPECjbb2000 with full occupancy.*

System	Average score (learning incl.)	Average score (learning excl.)
JRockit	22642,86	23293,98
RLS	23093,08	24775,43
Improvement (%)	1,98832	6,359799

8 Discussion and Future Developments

The preliminary results of our study indicate that reinforcement learning might improve existing garbage collection techniques. However, a more thorough analysis and extended benchmark tests are required for an objective evaluation of the potential of reinforcement learning techniques for dynamic memory management.

The most important task of future investigation is to systematically investigate the effect of using additional state features for the decision process and to investigate their usefulness for making better decisions.

The second important aspect is to investigate more complex scenarios of memory allocation, in which the memory allocation behavior switches more rapidly and less regularly. It is also of interest to investigate other dimensions of the garbage-collection problem such as object size and levels of references between objects, among others. It is important to emphasize that the results above are derived from a limited set of test applications that cannot adequately represent the range of all possible applications.

The issue of selecting proper test application environments also relates to the problem of generalization. The question is: how much does training on one particular application or a set of multiple applications help to perform well on unseen applications? It would be interesting to investigate how long it takes to learn from scratch or how fast an RLS can adapt when the application changes dynamically.

Another suggestion for improving the system is to decrease the learning rate more slowly. The same suggestion applies to the probability of choosing a random action in order to achieve a better balance between exploitation and exploration. The optimal parameters are best determined by cross-validation.

An approach for achieving better results when more state features are taken into account might be to represent the state features in a different way. For instance, radial basis functions, mentioned earlier in this report, might be of interest for generalization of continuous state features. An even better approach would be to represent the state features with continuous values and to use a gradient-descent method for approximating the Q-function.

It seems that that the total number of state features is a crucial factor. JRockit considers only one parameter for the decision of when to garbage collect. The performance of the RLS was not improved using two state features, likely due to the enlarged state space. The question remains, whether the performance of the RLS improves if additional state information is available and the time for exploration is increased. The potential strength of the RLS might reveal itself better if the decision is based on more state features than JRockit uses currently.

Another important aspect is online vs. offline performance. How much learning can be afforded, or shall only online-performance be considered? That of course is also a design issue for JRockit, which relies on a more precise definition of the concrete objectives and requirements of a dynamic Java Virtual Machine.

Once a real system has been developed from the prototype, it can be used to handle some of the other decisions related to garbage collection proposed in this report.

It is recommended to investigate this research area further, since it is far from exhausted. Considering that the results were achieved using a prototype that is poorly adjusted in several aspects, further development might lead to interesting and even better results than obtained within the restricted scope of this project.

9 Conclusions

The trade-off that every garbage collecting system faces is that garbage collection in itself is undesirable, as it consumes time from the running program. However, if garbage collection is not performed the system runs the risk of running out of memory, which is far worse than slowing down the application. The motivation for using a reinforcement learning system is to optimize this trade-off between saving CPU time and avoiding exhaustion of the memory.

This report has investigated how to design and implement a learning decision process for a more dynamic garbage collection in a modern JVM. The results of this thesis show that it is in principle possible for a reinforcement learning system to learn when to garbage collect. It has also been demonstrated that on simple test cases the performance of the RLS after training in terms of the reward function is comparable with the heuristics of a modern JVM, such as JRockit.

The time it takes for the RLS to learn also seems reasonable since the system only runs out of memory 5-10 times during the learning period. Whether this cost of learning a garbage collecting policy is acceptable in real applications depends on the environment and the requirements on the JVM.

From the results in the case of two state features, it becomes clear that using multiple state features potentially results in more complex decision surfaces than simple standard heuristics. Observations have also been made that there exists an evident trade-off between using more state features, in order to make more optimal decisions, and the increased time required for learning due to an enlarged state space.

From the above results one can learn that the use of a reinforcement learning system is particularly useful if an application has a complex dynamic memory allocation behavior, which is why a dynamic garbage collector was proposed in the first place. It is noteworthy to observe that machine learning through an adaptive and optimizing decision process can replace a human designed heuristic such as JRockit that operates with a dynamic threshold.

This article is an excerpt of the project report *Reinforcement Learning for a Dynamic JVM* [6], which may be obtained by contacting the author at: *eva.andreasson@appeal.se*.

10 References

Literature

1. Bertsekas, D. P. and Tsitsiklis, J. N. (1996). *Neuro-dynamic programming*. Athena Scientific, Belmont, Massachusetts, USA.

2. Jones, R. and Lins, R. (1996). *Garbage collection – algorithms for automatic dynamic memory management*. John Wiley & Sons Ltd., Chichester, England, UK.

3. Mitchell, T. M. (1997). *Machine learning*. McGraw Hill, USA.

4. Russell, S. J. and Norvig, P. (1995). *Artificial intelligence – a modern approach*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, USA.

5. Sutton, R. S. and Barto, A. G. (1998). *Reinforcement learning – an introduction*. MIT Press, Cambridge, Massachusetts, USA.

Papers

6. Andreasson, E. (2002). *Reinforcement Learning for a dynamic JVM*. KTH/Appeal Virtual Machines, Stockholm, Sweden.

7. Brecht, T., Arjomandi, E., Li, C. and Pham, H. (2001). *Controlling garbage collection and heap growth to reduce the execution time of java applications*. ACM Conference, OOPSLA, Tampa, Florida, USA.

8. Flood, C. H. and Detlefs, D.; Shavit, N.; Zhang, X. (2001). *Parallel garbage collection for shared memory multiprocessors*. Sun Microsystems Laboratories, USA; Tel-Aviv University, Israel; Harvard University, USA.

9. Lindholm, D. and Joelson, M. (2001). *Garbage collectors in JRockit 2.2*. Appeal Virtual Machines, Stockholm, Sweden. Confidential.

10. Pack Kaelbling, L.; Littman, M. L. and Moore, A. W. (1996). *Reinforcement Learning: A Survey*. Journal of Artificial Intelligence Research, Volume 4.

11. Pérez-Uribe, A. and Sanchez, E. (1999). *A comparison of reinforcement learning with eligibility traces and integrated learning, planning and reacting*. Concurrent Systems Engineering Series, Vol. 54, IOS Press, Amsterdam.

12. Precup, D., Sutton, R. S. and Dasgupta, S. (2001). *Off-policy temporal-difference learning with function approximation*. School of computer science, McGill University, Montreal, Quebec, Canada and AT & T Shannon laboratory, New Jersey, USA.

13. Printezis, T. (2001). *Hot-swapping between a mark&sweep and a mark&compact garbage collector in a generational environment*. Department of Computing Science, University of Glasgow, Glasgow, Scotland, UK.

14. Printezis, T.; Detlefs, D. (1998). *A generational mostly-concurrent garbage collector*. Department of Computing Science, University of Glasgow, Glasgow, Scotland, UK; Sun Microsystems Laboratories East, Massachusetts, USA.

15. Tsitsiklis, J. N. and Van Roy, B. (1997). *An analysis of temporal-difference learning with function approximation*. Laboratory for information and decision systems, MIT, Cambridge, Massachusetts, USA.

Optimizing Precision Overhead for x86 Processors

Takeshi Ogasawara   Hideaki Komatsu   Toshio Nakatani

IBM Japan, Tokyo Research Laboratory
1623-14 Shimotsuruma, Yamato-Shi, Kanagawa, Japan 242-8502
{takeshi,komatsu,nakatani}@jp.ibm.com

Abstract

It is a major challenge for a Java JIT compiler to perform single-precision floating-point operations efficiently for the x86 processors. In previous research, the double-precision mode is set as the default precision mode when methods are invoked. This introduces redundant mode switches to preserve the default precision mode across method boundaries. Furthermore, the precision mode is switched only at the start and end points of a given method to reduce the overhead of rounding double-precision results to single-precision ones. Therefore, methods that include both single- and double-precision operations cannot switch the mode and have to suffer from the overhead of rounding using this convention, even if single-precision operations are dominant.

We propose a new approach to these problems. We eliminate redundant mode switches by ignoring the default precision mode and calling a method in the same precision mode as the caller. For methods that include both single- and double-precision methods, we reduce the overhead of rounding by isolating code segments of a given method that should be executed in the single-precision mode. We implemented our approach in IBM's production-quality Just-in-Time compiler, and obtained experimental results, showing that, in SPECjvm98, it consistently shows the best performance in any configuration of benchmark programs, inline policies, and processor architectures compared to previous research.

1 Introduction

The Java language [4] has unique specifications that enable programmers to create highly portable programs. In particular, it requires floating-point operations that conform to the IEEE 754 standard [5]. Under these specifications, Java programs running on a different platform can calculate the same result for a given floating-point operation. FP-strictness [21] as introduced by Java 2 has relaxed the specification with respect to overflow and underflow, but it still calls for strict precision.

A Just-in-Time (JIT) compiler for Java [15, 3, 17, 19, 16, 2] is critical for high-performance Java virtual machines. It must follow the language specification, while it is also expected to generate efficient code. A JIT compiler for the x86 processors faces several performance challenges unique to the x86 architecture.

In particular, it is a major challenge for a Java JIT compiler to perform single-precision floating-point operations efficiently for the x86 processors. Unlike other processors, the x86 processor has a precision mode that determines whether the floating-point unit executes the same instruction in either single or double precision. In general, the double precision mode is usually set as the default precision mode throughout the program execution, and the single-precision floating-point operations are emulated in the double-precision mode. This is true even for recent processors that support a new instruction set, the Streaming Single-instruction-multiple-data Extensions 2 (SSE2) [8], since the original instruction set can perform some operations more efficiently.

There are two approaches to performing single-precision floating-point operations for Java programs in the double-precision mode of the x86 processors. One [12] is to emulate the single-precision floating-point operations in the double-precision mode by rounding a double-precision result to the single-precision value. This can be accomplished by storing it from the register to a single-precision memory and loading it back again to the register. This store-reload for rounding is required for each single-precision floating-point operation, because Java doesn't allow the double-precision values to be accumulated for the conversion to a single-precision value once at the end. Obviously the drawback of this store-reload approach is the latency of store-to-load forwarding [13] for each single-precision floating-point operation.

The other approach is to switch the mode from the double precision to the single precision mode at the start point of the method and perform all the floating-point operations in the single-precision mode without emulation. The drawback of this mode-switch approach is the penalty of mode switches at every point of the code where the control is transferred from and to another code block. Furthermore, this approach is not applicable to those methods that include both double-precision and single-precision floating-point operations, since obviously a double-precision operation cannot be performed in the single-precision mode.

Recently Paleczny et al. [17] proposed a mixed approach in which the mode switch is used whenever it gains performance over the store-reload approach. This eliminates the overhead of some cases for the store-reload approach, but it still suffers from the overhead of the mode switch and a high overhead for those methods that include both double-precision and single-precision floating-point operations.

In this paper, we propose a new approach called *precision-aware invocation*, in which we ignore the default precision mode and call a method in the same precision mode as the caller whenever appropriate. We add a property called *floating-point precision type (FPPT)* to each compiled code block. In our approach, we eliminate the overhead of redundant switching between the single-precision mode and the default precision mode (that is, double-precision mode) by preserving the default precision mode across the method boundaries. Only when we call a method of a different precision mode must we switch the precision mode at the start and end points of the method.

We further propose a technique called *precision region analysis* to reduce the number of operations to emulate single-precision floating-point operations in the double-precision mode using the store-reload approach. In this technique, our JIT compiler analyzes the bytecode of the given methods, and locates the places at which it should switch the precision mode at a finer granularity without switching the mode only at the start and end points of the method.

In summary, the contributions of our paper are the following:

- We minimize the number of redundant mode switches in the JIT compiled code by eliminating the default precision mode and introducing precision-aware invocation based on the floating-point precision type of each method.
- We reduce the number of operations to emulate single-precision floating-point operations in the double-precision mode by doing precision region analysis in the JIT compiler.

The rest of this paper is organized as follows. Section 2 describes the portability of Java floating-point types and the overhead of executing single-precision floating-point operations on the x86 processors. Section 3 describes our new techniques to minimize the redundant mode switches and the number of operations to emulate them in the double-precision mode. Section 4 presents our experimental results. Section 5 discusses related work. Finally, Section 6 concludes the paper.

2 Portability and Overhead of the Java Floating-Point Types

This section describes the Java specification for floating-point operations and how single-precision floating-point operations are performed in conforming to it on the x86 processors. The section then describes what problems previous JIT compilers for the x86 face in order to perform these operations efficiently.

2.1 Java Floating-Point Operations and the IEEE 754 Standard

Java has two floating-point types: float and double. Floating-point values of these types and operations on them should conform to the IEEE 754 standard [5, 22]. The standard defines the binary floating-point format, which consists of three components: sign, exponent, and significand. The Java floating-point types, float and double, are associated with the single and double precision formats of the standard, respectively. Therefore, all the Java virtual machines should generate the same result for a given floating-point expression at the bitwise level in their binary formats.

Java 2 has introduced a new semantic category, FP-strict [21]. The semantics of FP-strict code are the same as that of the original specifications. On the other hand, non-FP-strict code still performs a floating-point operation with the same size of significand as that of FP-strict code, but it extends the exponent. Regardless of FP-strictness, it is not permissible to perform a single-precision operation in double precision.

2.2 Floating-Point Unit and Precision Mode on the x86

The x86 has a floating-point unit that supports the IEEE 754 standard [9]. The floating-point unit does not have separate instructions for each type of floating-point precision. Instead, it executes an instruction in either single or double precision by switching the precision mode. There is a special instruction, fldcw, which modifies the hardware flags that switch the mode. However, since the overhead of fldcw is quite large, frequent switching of the precision mode degrades the performance of a program. Although the overhead of fldcw is optimized in the NetBurst [6], it is still a significant penalty.

There is a way to perform a single-precision operation so that a program does not suffer from the penalty of switching the precision mode. First, perform the operation in double precision. Second, store the calculated result in the register into a memory location of single-precision size. This translates the result to the single-precision value. Third, load it back into the register. However this rounding store-reload operation [12] inserts additional latency in the data flow, even though the store-to-load forwarding feature enables

```
                     ; start in double-precision mode
fldcw [single_prec]  ; switch to single-precision mode
  :
fmul  ST(0), ST(0)   ; r0 = r0*r0
faddp ST(1), ST(0)   ; r1 = r1+r0
  :
fldcw [double_prec]  ; switch to double-precision mode
ret                  ; return in double-precision mode
```

(a) The mode switch method

```
                     ; start in double-precision mode
  :
fmul  ST(0), ST(0)   ; r0 = r0*r0
fstp  real4 ptr [mem]; store r0 to mem (rounding)
fld   real4 ptr [mem]; load r0 from mem
faddp ST(1), ST(0)   ; r1 = r1+r0
fxch  ST(0), ST(1)   ; exchange r0 with r1
fstp  real4 ptr [mem]; store r0 to mem (rounding)
fld   real4 ptr [mem]; load r0 from mem
  :
ret                  ; return in double-precision mode
```

(b) The rounding store-load method

Figure 1: Examples of x86 single-precision operations

the load to receive the stored value from the store buffer without accessing memory. Figure 1 shows two versions of x86 instructions for these two methods, which perform the same single-precision operations, $v1 = v1 + v2 * v2$.

2.3 Problems of Previous JIT Compilers

For Java Just-in-Time (JIT) compilers for x86-based processors [3, 17, 19], it is critical to reduce the precision-related overhead to optimize a Java program that frequently executes floating-point operations. In previous research, double precision is used as the default mode for compiled code [17]. This default precision mode was required because the interpreter usually runs in the double precision mode and selectively compiled code can be called by an interpreted method or by a method that will be dynamically loaded in the future.

If a traditional compiler analyzes a method and the analysis shows that the improvement by eliminating the rounding store-reloads exceeds the overhead of executing fldcw [17], it inserts an fldcw instruction to start the code in the single precision mode. It also inserts an fldcw to restore the default mode at each exit point. If this method invokes another method that is also analyzed to switch to the single precision mode, the compiler inserts an fldcw to restore the default mode before the invocation and another fldcw to switch to the single precision mode after the invocation. There are variations of this approach with different thresholds for determining whether or not to switch the mode. This approach balances the cost of fldcw and the latency caused by the rounding store-reload, while it minimizes the total penalty for supporting Java's floating-point specification. However, it still suffers from the following two forms of overhead.

The first overhead is redundant switches of the precision mode during method invocations. Consider the case when a method runs in single precision and invokes another method that prefers to run in single precision. Since the default mode is double and every method assumes that it is invoked in double precision, the caller method executes an fldcw to switch the precision mode from single to double before the invocation. Then the callee method immediately executes a second fldcw to switch from double to single. A similar redundant switch occurs when the callee method returns to the caller.

The second overhead is the rounding store-reloads caused by the convention of the mode switch required at the start and end points. Since default-precision operations must be performed in the double-precision mode, this convention makes the entire code run in the double-precision mode if the code includes double-precision operations as well as single-precision operations. As a result, these single-operations require rounding store-reloads. Another case is when a compiler determines that the penalty of these rounding store-reloads is less costly than that of the mode switch at the start and end points even if the code includes single-precision operations but not double-precision operations. The problem is that, if these rounding store-reloads are executed frequently at runtime, the cumulative overhead significantly degrades the performance of the program.

These overheads for a program that performs single-precision operations are caused by the default precision mode and the ordinary mode-switch convention. To minimize these overheads, we need to analyze the entire call graph of methods to determine where the precision mode changes. However, it is difficult for a JIT compiler to do such an analysis because it does not compile all the methods at one time. In particular, JIT compilers cannot analyze interpreted methods when they perform selective compilation to reduce the time and resources for compilation [17, 19]. Our goal is to remove these overheads to conform to the IEEE 754 standard without analyzing the entire program.

3 Precision Region Optimization (PRO)

This section describes our new techniques to minimize redundant mode switches and reduce store-reload operations for rounding in those methods which include both single- and double-precision operations.

3.1 Tracking the Floating-Point Precision Type of Code Blocks

To eliminate redundant mode switches and reduce the memory latency of rounding store-reloads, our JIT-compiled code has an attribute called *floating-point precision type (FPPT)*. The FPPT of code shows the precision mode in which the code is called. When a method calls another method, it calls the code with the same FPPT as the current precision of the call site.

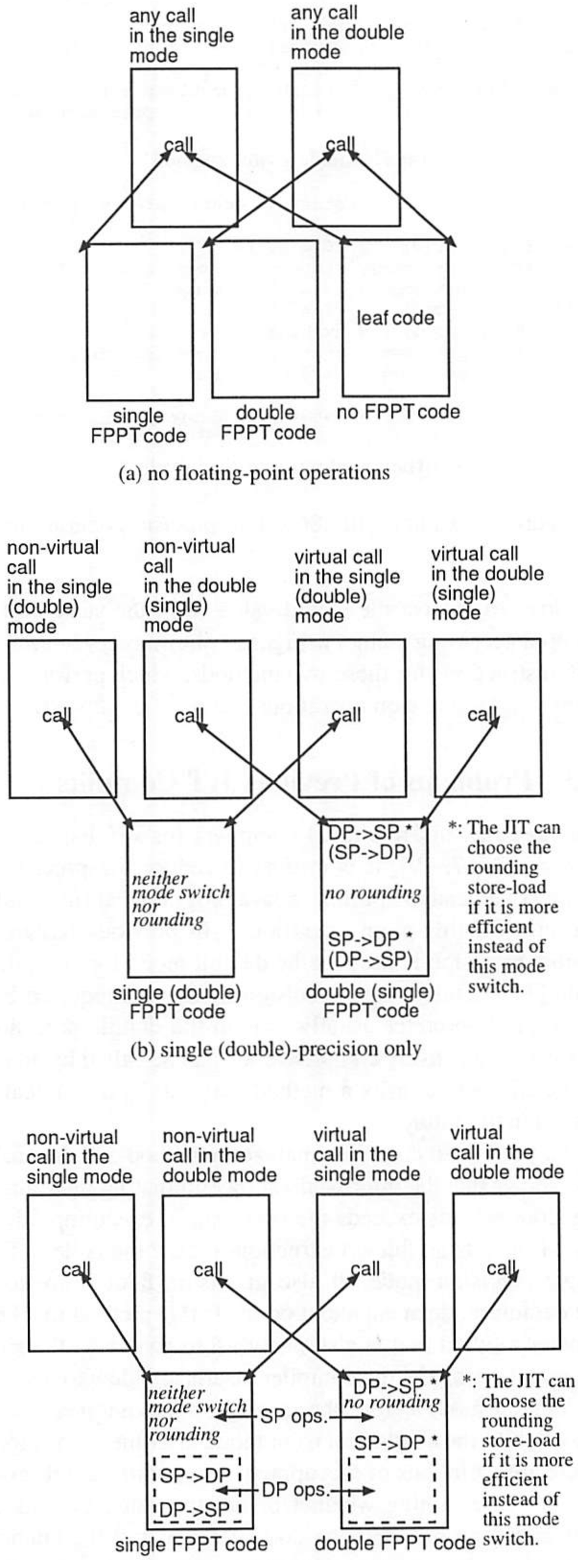
Figure 2 shows all of the combinations of caller's precision (single or double), the type of invocations (virtual or non-virtual), callee's FPPT (single or double), and the type of floating-point operations included in the code (none, single, double, or both). Each rectangle shows a code segment. DP and SP denote double and single precision, respectively. Each arrow shows a method invocation. The code for an FPPT is generated only if a method invocation that points to it actually occurs.

We explain how we generate the code for an FPPT for three different cases using Figure 2. The first case is when calling a method whose bytecode does not include any floating-point operations (Figure 2a). If it includes call sites, the compiled code has a specific FPPT since it had to determine the FPPT for the callee code. The body of the single-FPPT code block and that of the double-FPPT are the same. If the method is a leaf method, the FPPT is not used. The second case is when calling a method whose bytecode includes floating-point operations of either single or double precision (Figure 2b). There is no mode switch if the caller's precision is the same as the precision of these operations. Otherwise, the compiler analyzes the bytecode to determine whether the mode switch is required, considering the tradeoff. For the mode switch, it is performed only once across the method boundary. In addition, since the code bodies for the same-precision and different-precision cases are the same, the compiler can easily generate one by duplicating the other. The third case is when calling a method whose bytecode includes single and double operations (Figure 2c). This case is the same as the second one except that switching to double occurs to perform double-precision operations in the single-FPPT code.

The binary translation from the code of one FPPT to that of the other FPPT is done simply by inserting or removing instructions for the mode switch and the rounding store-reload. Since this is much faster than the compiler's optimizations for the original code, the overhead of introducing the FPPT is negligible at compilation time. However, in general, a method tends to be called from either single or double precision. Therefore, it is expected that the majority of methods have either a single- or a double-FPPT version of the code and the degree of code expansion is quite low.

3.2 Precision Region Analysis

If a method includes both single- and double-precision operations (the case of Figure 2c), the approach of previous research makes the compiled code run in the double-precision mode because double-precision operations must be performed in the double-precision mode. However, this introduces the penalty of extra rounding store-reloads for the single-precision operations. In particular, this problem incurs a significant penalty when a method includes many single-precision operations but only a few double-precision operations.


any call in the single mode
any call in the double mode
call
call
leaf code
single FPPT code
double FPPT code
no FPPT code

(a) no floating-point operations

non-virtual call in the single (double) mode
non-virtual call in the double (single) mode
virtual call in the single (double) mode
virtual call in the double (single) mode
call
call
call
call
neither mode switch nor rounding
DP->SP *
(SP->DP)
no rounding
SP->DP *
(DP->SP)
*: The JIT can choose the rounding store-load if it is more efficient instead of this mode switch.
single (double) FPPT code
double (single) FPPT code

(b) single (double)-precision only

non-virtual call in the single mode
non-virtual call in the double mode
virtual call in the single mode
virtual call in the double mode
call
call
call
call
neither mode switch nor rounding
SP ops.
DP->SP *
no rounding
SP->DP *
SP->DP
DP->SP
DP ops.
*: The JIT can choose the rounding store-load if it is more efficient instead of this mode switch.
single FPPT code
double FPPT code

(c) single and double operations

Figure 2: Precision-aware method invocations and code generation.

The JIT compiler can determine which code segments perform single-precision operations in the single-precision mode rather than making the entire code block run in the double-precision mode. The basic idea is to *shrink wrap* program regions that run in the double-precision mode. Since there is a performance trade-off between the mode switch method and the rounding store-reload method, this shrink-wrapping is relaxed for a double-precision region and single-precision operations adjacent to the region may be included in the region to minimize the overhead of calculating single-precision results.

Figure 3 shows the algorithm of our *precision region analysis*. The first step of the algorithm is the *intra* basic block analysis. The JIT compiler sections each basic block so that each code section includes either single- or double-precision operations. It also counts the number of floating-point operations for each section. The second step is the *inter* basic block analysis. The JIT compiler arranges basic blocks in depth-first-search order based on the control flow graph, considering the history of the branches recorded while the interpreter executed the method. The algorithm processes basic blocks in this order. The result is that it prioritizes the optimization of a control path that has been frequently executed by the interpreter, since the target basic block of a frequently taken branch appears earlier than that of a rarely taken branch in this order. The JIT compiler traverses basic blocks in this order and checks the precision of each section. For contiguous single-precision sections, it records entering them and counts the number of operations. This recording of single-precision sections continues until the JIT compiler encounters a double-precision section, the head of a loop, or a basic block that is not a target of the previous basic block. When there are enough single-precision operations to justify switching the mode, the precision of these sections remains as single. Otherwise, they are modified to be double-precision sections, so that the code of these sections runs in the double-precision mode. For the basic blocks of exception handlers, the JIT compiler can propagate the precision of an exception-throwing point to its handler, although this process is not shown in Figure 3.

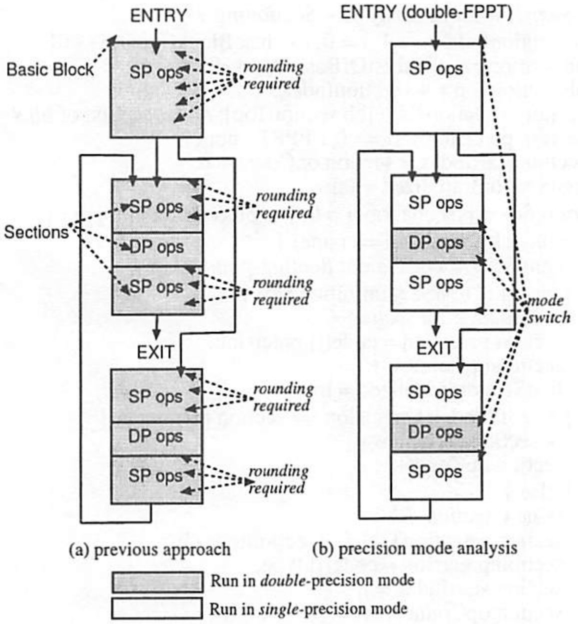
Figure 4 illustrates how floating-point operations are performed when a method includes both single- and double-precision operations by using an example. Figures 4a and 4b show the results of the previous approach and our precision region analysis, respectively. In Figure 4a, all of the sections are in a double-precision region. By applying the precision region analysis, only double-precision sections are shrink-wrapped with mode-switch instructions in Figure 4b. In this example, the algorithm finds out that the overhead of rounding store-reloads required in the first and second sections is higher than the cost of two mode switches. As a result, these sections are marked as single-precision so that the code generation generates mode-switch instructions. Similarly, it determines that the fourth, fifth, and eighth sections should run in the single-

```
/* Step 1: Intra-bb analysis – Sectioning */
for (sectionIndex = −1, i = 0; i < bsicBlockCount; i++) {
  bb = prioritizedDfsListOfBasicBlocks[i];
  bb.sectionTop = ++sectionIndex;
  section = sectionTable[bb.sectionTop]; /* 1st section of bb */
  section.precision = (i == 0 ? FPPT : none);
  section.startIndex = section.opCount = 0;
  firstSectionInitialized = false;
  for (code = bb.codeTop, j = 0; j < bb.codeLength; j++) {
    if (code[j].precision == none) {
      continue; /* code is not floating-point; skip */
    } else if (! firstSectionInitialized) {
      /* initialize 1st section */
      section.precision = code[j].precision;
      section.opCount = 1;
      firstSectionInitialized = true;
    } else if (code[j].precision == section.precision) {
      /* section continues */
      section.opCount++;
    } else {
      /* new section */
      section = sectionTable[++sectionIndex];
      section.precision = code[j].type;
      section.startIndex = j;
      section.opCount = 1;
    }
  } /* end for j */
  /* bb consists of only 1st section if 1st section is uninitialized */
  bb.sectionCount = sectionIndex − bb.sectionTop + 1;
}

/* Step 2: Inter-bb analysis – Minimizing SP calculation cost */
pendingSpSectionOpCount = 0;
pendingSpSectionIndex = −1;
for (i=0, prevBB=null; i < basicBlockCount; i++, prevBB=bb) {
  bb = prioritizedDfsListOfBasicBlocks[i];
  if (pendingSpSectionIndex ≥ 0 &&
      (bb.isLoopHead || ! bb.isBranchedFrom(prevBB))) {
    /* process before a loop or non-contiguous bb */
    processPendingSpSections();
  }
  section = sectionTable[bb.sectionTop];
  for (k = 0; k < bb.sectionCount; k++) {
    switch (section[k].precision) {
    case SP:
      /* record potential-SP sections */
      pendingSpSection[++pendingSpSectionIndex] = section[k];
      pendingSpSectionOpCount += section[k].opCount;
      break;
    case DP:
      if (pendingSpSectionIndex ≥ 0) {
        /* process at boundary from SP to DP */
        processPendingSpSections();
      }
    }
  } /* end for k */
} /* end for i */

/* Subroutine: Process pending SP sections */
processPendingSpSections() {
  /* roundCost and modeSwitchCost are machine-specific const */
  if (pendingSpSectionOpCount*roundCost<modeSwitchCost) {
    /* mode-switch cost is higher; make the sections run in DP */
    for (m = 0; m ≤ pendingSpSectionIndex; m++)
      pendingSpSection[m].precision = DP;
    pendingSpSectionIndex = −1;
    pendingSpSectionOpCount = 0;
  }
}
```

Figure 3: The algorithm of precision region analysis



(a) previous approach (b) precision mode analysis

Figure 4: An example of shrink-wrapping double-precision regions

precision mode.

3.3 Precision-Aware Invocation

For each call site within the method, the JIT compiler uses the precision mode at the call site and generates instructions that call the target code of the FPPT that matches that precision mode. The method block that maintains the information for a method has two entries for code pointers, one for single-FPPT and another for double-FPPT. The generated instructions use one of these pointers, depending on the precision mode at the call site.

4 Measurements

In this section, we evaluate precision-aware invocation using our system. We used the modules of SPECjvm98 as benchmark programs. Subsection 4.1 explains the methodology of our evaluation. Subsection 4.2 explains the system used in our experiments. Subsection 4.3 explains the details of the benchmark programs used in our experiments, focusing on the floating-point operations. Subsection 4.4 presents the results and discusses them.

4.1 Experimental Methodology

For practicality, we used the SPECjvm98 benchmark suite [23] in the test mode throughout our experiments. SPECjvm98 is a suite of benchmark programs and is currently accepted as one of the major Java benchmarks for evaluating Java VMs [16, 19, 2, 3].

We compare the rounding store-reload (denoted as Rounding), the mode-switch method (denoted as Switch), previous research [17] (denoted as HSS), and our precision region optimization (denoted as PRO) with each other. We use Roundign and Switch to emulate the two extreme conventions with respect to controlling the precision mode. In Rounding, single-precision operations are always performed in the default precision mode, or double. This emulates the ordinary approach of ignoring the penalty of rounding store-reloads. Therefore, it can be used to measure the maximum number of possible rounding store-reloads. In Switch, single-precision operations are always performed in the single-precision mode by switching the mode at the start and end points if the method includes only single-precision operations. Code that includes a double-precision operation always runs in the double-precision mode. This emulates the ordinary approach of ignoring the effects of the numbers of call sites. Therefore, this can be used to measure the maximum number of possible mode switches. In HSS, the Switch approach is applied for methods that include enough single-precision operations (at least 32 and more than 10 times of the number of call sites), but not any double-precision operations, and the Rounding approach is applied for other methods.

4.2 Environment

We implemented precision region optimization on our production quality Java Just-In-Time (JIT) compiler [18, 11, 10, 14, 19], which is part of the IBM Developer Kit for Windows, Java 2 Technology Edition [7], Version 1.3.1.

Throughout the measurements, we used the same parameters for the Java VM and SPECjvm98. The initial and maximum amounts of Java heap space were 128 MB, specified with the parameters `-Xms128m -Xmx128m`. The SPECjvm98 parameter string `-m5 -M5 -s100` causes each benchmark program to be executed five times using a problem size of 100.

The measurements were performed on an Intel Pentium-III 1GHz CPU with 512 MB physical memory running Microsoft Windows NT Workstation Version 4.0 with Service Pack 6. For measurements of the execution time with other examples of the x86 architecture, we used an AMD Athlon MP 1.2GHz processor and an Intel Pentium 4 2.0GHz processor.

4.3 Characteristics of the Benchmark Programs

This section gives an overview of the characteristics of the benchmark programs. We focus on two benchmark programs of SPECjvm98 that are floating-point intensive. The other programs are not. Table 1 shows the number for each category of floating-point operations for each benchmark program of SPECjvm98 during the fifth run, by which

benchmark	XPO[1]	SP arith	DP arith	Compare	Global ld/st	Prec Conv	Int Conv
_227_mtrt	off	114,575,305	193,540	52,202,408	230,989,486	386,490	132,001
	on	114,768,845	0	52,202,408	230,989,486	8	132,001
_222_mpegaudio	off	1,091,260,405	1,549,221	40,112,640	781,219,540	1,549,221	20,065,036
	on	1,092,809,573	53	40,112,640	781,219,540	53	20,065,036

Table 1: The dynamic counts of floating-point operations

		no inlining			tiny inlining			aggressive inlining		
		total	w/SP		total	w/SP		total	w/SP	
benchmarks	XPO		w/o DP	w/DP		w/o DP	w/DP		w/o DP	w/DP
_227_mtrt	off	310,344,583	5.11%	0.04%	22,810,014	52.11%	1.26%	11,738,192	38.98%	2.31%
_227_mtrt	on	310,344,583	5.19%	0.02%	22,810,016	53.15%	0.23%	11,738,192	40.85%	0.44%
_222_mpegaudio	off	117,631,096	3.95%	1.30%	36,768,517	12.64%	4.16%	21,874,285	19.54%	7.00%
_222_mpegaudio	on	117,631,096	4.61%	0.64%	36,768,517	14.75%	2.06%	21,874,285	23.08%	3.46%

Table 2: The dynamic counts of method invocations

benchmark	XPO	no inlining	tiny inlining	aggressive inlining
_227_mtrt	off	0.262% (4)	2.057% (8)	3.589% (9)
	on	0.071% (1)	0.071% (1)	0.084% (1)
_222_mpegaudio	off	3.226% (2)	3.226% (2)	3.226% (2)
	on	0.812% (1)	0.812% (1)	0.812% (1)

Table 3: The dynamic characteristics of single-precision operations performed by methods that include both single- and double-precision operations

time all of the major methods have been selectively compiled by the JIT compiler. Since these numbers are counted at the intermediate-language level but not at the machine-instruction level, they are independent of machine architectures. The XPO column is explained in the next paragraph. The SP arith and DP arith columns of Table 1 show the numbers of single- and double-precision arithmetic operations such as addition, respectively. The Compare column shows the number of compare operations [2]. The Global ld/st column shows the number of memory operations for the global area such as static fields and object fields. This does not include memory operations that required by spilling registers in and out for the local frame. The Prec Conv column shows the number of operations that convert a value of one precision to the other. The last column shows the number of operations that convert a floating-point value to the corresponding integer value.

The XPO column shows whether excessive-precision operations are optimized or not. In some cases, source operands for a double-precision operation have only single precision and the result of the operation is immediately converted into single precision. If the corresponding operation in single precision can calculate the same value as the double-precision operation does for these single-precision values, we can translate this excessive-precision operation into a single-precision one. As shown in Table 1, this excessive-precision optimization reduced the dynamic counts of double-precision operations.

The problem caused by a default precision mode across the method boundaries has been discussed in Section 2.3. The resulting performance degradation is proportional to the number of method invocations. Table 2 shows the number of method invocations during the fifth run for each level of *method inlining*. We applied three policies of inlining: no inlining, tiny method inlining, and aggressive inlining. In tiny method inlining, only tiny methods are inlined into the target code. In aggressive inlining, methods are inlined unless their calling depths and the code expansion reach the pre-defined thresholds. For each inlining policy, the invocation count of all the methods, that of the methods that include single-precision but not double-precision operations, and that of the methods that include both single- and double-precision operations during the fifth run are shown in Table 2. The number of redundant mode-switches is proportional to the invocation count of the methods that include single-precision but not double-precision operations.

We have also discussed the problem of the ordinary mode-switch convention in the case of methods that include both single- and double-precision operations in Section 2.3. Table 3 shows the dynamic count of single-precision operations performed by these methods and the

[1] XPO = excessive-precision optimization

[2] The x86 compares all of the 80 bits of floating-point values in the registers regardless of the precision mode.

		no inlining		tiny method inling		aggressive inling	
benchmarks		switch	store-reload	switch	store-reload	switch	store-reload
_227_mtrt	Rounding	0	82,591,461	0	81,921,296	0	73,964,450
	Switch	435,817,517	102,613	30,328,119	102,597	8,921,599	102,597
	HSS	0	82,591,461	0	81,921,296	1,894,731	36,379,374
	PRO	108,901	51,437	108,901	51,437	108,901	51,437
_222_mpegaudio	Rounding	0	983,089,608	0	983,072,178	0	983,072,178
	Switch	50,667,740	17,484	10,485,380	54	10,169,023	54
	HSS	9,322,377	642,969,130	9,322,377	642,951,700	9,322,377	642,951,700
	PRO	3,649,659	17,484	3,649,659	54	9,760	54

Table 4: The dynamic counts of mode switches and rounding store-reloads

ratio for the total floating-point operations in each benchmark program. In previous research, these single-precision operations have to be performed in double precision, so this table shows the maximum numbers of single-precision operations that precision region analysis can optimize with respect to the overhead of rounding store-reloads. For these benchmark programs, excessive-precision optimization reduces the number of methods that include both single- and double-precision operations by translating the double-precision operations into the single-precision ones. However, there still remains a small number of single-precision operations that require rounding store-reloads.

4.4 Results and discussion

In this section, we assume that excessive-precision optimization has been applied.

4.4.1 Dynamic counts of mode switches and rounding store-reloads

Table 4 shows the number of mode switches and rounding store-reloads during the fifth run for each method-inlining policy. Overall, PRO significantly improve these dynamic counts for each policy of inlining. The store-reload columns for PRO show the numbers of the type conversion operations performed in every configuration. In addition to them, a method that has to run in double precision because of including both single- and double-precision operations performs rounding store-reloads for single-precision operations in Switch for _227_mtrt.

4.4.2 Execution time

Tables 5 through 7 show the execution time of the benchmark programs in seconds using Rounding, Switch, HSS, and PRO for each method-inlining policy. Overall, PRO consistently shows the best performance in any configuration of inline policies and processor architectures. On the other hand, the best performer among Rounding, Switch, and HSS changes depending on benchmark programs, inline policies, and processor architectures. For instance, in _227_mtrt, Switch shows the worst performance without inlining, though it is the best of the three with tiny and aggressive inlining on Pentium 4 and Athlon MP. In _222_mpegaudio, Switch shows the best performance of the three even without inlining. Most importantly, PRO with tiny inlining shows performance comparable to the best performance of the others with aggressive inlining in every configuration of benchmark programs and processor architectures. This means that PRO can greatly improve the performance of the compiled code during the code optimization stage without incuring the heavy compilation overhead caused by aggressive inlining [20].

5 Related Work

A report of the Java Grande Forum [12] discusses the problems that arise when making Java programs that frequently perform floating-point operations run efficiently on the x86 processors while conforming to the Java specification. It describes a technique for another x86-specific problem, or double rounding, as well as the rounding store-reload technique. We did not address this problem in this paper.

Another previous research report [17] presents a heuristic for determining when to insert fldcw instructions by using the number of single-precision operations, the number of double-precision operations, and the number of call sites. This approach suffers from the overhead of redundant mode switches across the method boundaries. In addition, there are problems from the overhead of rounding store-reloads performed by methods that include both single- and double-precision operations.

Streaming SIMD Extensions 2 (SSE2) [6] introduced a new set of floating-point instructions to the x86. It has new instructions that operate on separate floating-point registers. There are some differences between the SSE2 and the original instruction set. The SSE2 instructions do not completely correspond to the original instructions, and some instructions do not exist in SSE2. In addition, the size of single-precision registers is half of that of the double-precision registers, while the original registers can contain both single- and double-precision values. Furthermore, the

benchmarks		no inlining	tiny method inlining	aggressive inlining
_227_mtrt	Rounding	4.313	2.719	2.406
	Switch	10.578	3.140	2.391
	HSS	4.312	2.703	2.250
	PRO	3.969	2.375	2.062
_222_mpegaudio	Rounding	7.687	7.203	7.062
	Switch	6.734	5.563	5.421
	HSS	7.234	6.750	6.625
	PRO	5.826	5.453	5.219

Table 5: The execution times on Pentium III

benchmarks		no inlining	tiny method inlining	aggressive inlining
_227_mtrt	Rounding	3.105	2.193	2.063
	Switch	4.527	1.942	1.732
	HSS	3.164	2.193	1.843
	PRO	2.774	1.793	1.663
_222_mpegaudio	Rounding	5.809	5.377	5.408
	Switch	3.665	3.155	3.175
	HSS	4.977	4.536	4.537
	PRO	3.565	3.134	3.034

Table 6: The execution times on Pentium 4

benchmarks		no inlining	tiny method inling	aggressive inlining
_227_mtrt	Rounding	3.535	2.503	2.273
	Switch	6.149	2.473	2.133
	HSS	3.545	2.483	2.153
	PRO	3.175	2.213	1.983
_222_mpegaudio	Rounding	6.319	5.818	5.828
	Switch	4.586	3.895	3.906
	HSS	5.598	5.087	5.107
	PRO	4.296	3.835	3.765

Table 7: The execution times on Athlon MP

original instruction set can perform more efficiently than SSE2 for some Java floating-point operations. Thus, since SSE2 has different characteristics from the original instruction set, there are situations where one or the other system is superior.

6 Conclusions

This paper has presented a novel approach to optimization of Java floating-point operations for the x86. It consists of tracking the floating-point precision type of code blocks, precision region analysis, and precision-aware invocation. By generating the code blocks with the appropriate floating point precision types, the default precision mode can be ignored. Precision region analysis investigates the code and finds appropriate code points where mode switch instructions can be inserted to minimize the overhead of rounding store-reloads. Finally, since the compiled code calls the target code with the same floating-point precision type as the call site, there is no redundant mode switch across the method boundaries. Our new approach does not sacrifice the strictness of the precision, while it minimizes the x86-specific overhead to preserve the strictness.

We have presented experimental data about the dynamic characteristics of floating-point operations, using floating-point intensive programs from a widely used benchmark suite. Using that data, we have shown that these programs perform many rounding store-reloads and mode switches, as well as the floating-point operations. We have presented experimental results with a modified version of IBM's production-level Just-in-Time compiler and discussed the effectiveness of our approach. Experimental results show that, in floating-point intensive programs, our approach greatly reduced rounding store-reloads and redundant mode switches and consistently shows the best performance in any configuration of benchmark programs, inline policies, and x86 processor implementation.

References

[1] *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-2000)* (New York, NY, USA, 2000), ACM Press.

[2] ARNOLD, M., FINK, S., GROVE, D., HIND, M., AND SWEENEY, P. F. Adaptive optimization in the Jalapeño JVM. In ACM [1], pp. 47–65.

[3] CIERNIAK, M., LUEH, G., AND STICHNOTH, J. M. Practicing JUDO: Java under dynamic optimizations. In *ACM SIGPLAN '00 Conference on Programming language design and implementation* (New York, NY, USA, May 2000), ACM Press, pp. 18–21.

[4] GOSLING, J., JOY, B., AND STEELE, G. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, Massachusetts, Aug. 1996.

[5] GOSLING, J., JOY, B., AND STEELE, G. *The Java Language Specification*. In *The Java Series* [4], Aug. 1996, ch. 4.2.3.

[6] HINTON, G., SAGER, D., UPTON, M., BOGGS, D., CARMEAN, D., KYKER, A., AND ROUSSEL, P. The microarchitecture of the pentium 4 processor. *Intel Technology Journal Q1* (2001).

[7] The IBM Developer Kit, Java 2 Technology Edition. http://www.ibm.com/developerworks/java/jdk/.

[8] INTEL CORPORATION. *P6 Family of Processors Hardware Developer's Manual*. Intel Corporation, Sept. 1998.

[9] INTEL CORPORATION. *IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*. Intel Corporation, Mt. Prospect, IL, 2001.

[10] ISHIZAKI, K., KAWAHITO, M., YASUE, T., KOMATSU, H., AND NAKATANI, T. A study of devirtualization techniques for a Java Just-In-Time compiler. In ACM [1], pp. 294–310.

[11] ISHIZAKI, K., KAWAHITO, M., YASUE, T., TAKEUCHI, M., OGASAWARA, T., SUGANUMA, T., ONODERA, T., KOMATSU, H., AND NAKATANI, T. Design, implementation, and evaluation of optimizations in a Java Just-In-Time compiler. *Concurrency: Practice and Experience 12*, 6 (2000), 457–475.

[12] Java Grande Forum Report: Making Java Work for High-End Computing. http://www.javagrande.org/sc98/sc98grande.pdf.

[13] JOHNSON, M. *Superscalar Microprocessor Design*. Prentice Hall Series in Innovative Technology. Prentice Hall, Englewood Cliffs, NJ, Jan. 1991.

[14] KAWAHITO, M., KOMATSU, H., AND NAKATANI, T. Effective null pointer check elimination utilizing hardware trap. In *Proceedings of the 9th international conference on Architectural support for programming languages and operating systems (ASPLOS-IX)* (New York, NY, USA, Nov. 2000), ACM Press, pp. 118–127.

[15] KRALL, A., AND PROBST, M. Monitors and exceptions: How to implement Java efficiently. In *ACM 1998 Workshop on Java for High-Performance Network Computing* (New York, NY, USA, 1998), ACM Press, pp. 15–24. Also published as *Concurrency: Practice and Experience*, **10**(11–13), September 1998, CODEN CPEXEI, ISSN 1040-3108.

[16] LEE, S., YANG, B.-S., KIM, S., PARK, S., MOON, S.-M., AND EBCIOĞLU, K. Efficient Java exception handling in Just-in-Time compilation. In *Proceedings of the ACM 2000 Conference on Java Grande* (New York, NY, USA, June 2000), ACM Press, pp. 1–8.

[17] PALECZNY, M., VICK, C., AND CLICK, C. The Java HotSpot server compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium* (Apr. 2001), USENIX Association, pp. 1–12.

[18] SUGANUMA, T., OGASAWARA, T., TAKEUCHI, M., YASUE, T., KAWAHITO, M., ISHIZAKI, K., KOMATSU, H., AND NAKATANI, T. Overview of the IBM Java Just-In-Time compiler. *IBM Syst. J. 39*, 1 (2000), 175–193.

[19] SUGANUMA, T., YASUE, T., KAWAHITO, M., KOMATSU, H., AND NAKATANI, T. A dynamic optimization framework for a Java Just-In-Time compiler. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-2001)* (New York, NY, USA, 2001), ACM Press, pp. 180–194.

[20] SUGANUMA, T., YASUE, T., AND NAKATANI, T. An empirical study of method inlining for a Java a Just-In-Time compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium* (Aug. 2002), USENIX Association.

[21] SUN. Updates to the Java language specification for JDK release 1.2 floating point. http://java.sun.com/docs/books/jls/strictfp-changes.pdf, Dec. 1998.

[22] THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC. IEEE standard for binary floating-point arithmetic, Oct. 1985.

[23] THE STANDARD PERFORMANCE EVALUATION CORPORATION (SPEC). JVM Client98 (SPECjvm98). http://www.spec.org/osg/jvm98/, 1998.

Experiences Porting the Jikes RVM to Linux/IA32

Bowen Alpern *Maria Butrico* *Anthony Cocchi*
Julian Dolby *Stephen Fink* *David Grove* *Ton Ngo*
IBM T. J. Watson Research Center
Yorktown Heights, NY, 10598

Abstract

This paper describes our experiences in porting the Jikes Research Virtual Machine from its first platform, AIX/PowerPC, to its second, Linux/IA32. We discuss the main issues in realizing both an initial functional port, and then tuning efforts to achieve competitive performance. The paper presents software engineering issues in building a portable runtime system and compilers, as well as specific optimizations to improve performance on IA32.

1 Introduction

In early 2001, developers of the Jikes^{TM} Research Virtual Machine (RVM) committed to releasing the software open-source by the end of 2001. At the time, the virtual machine ran on PowerPC^{TM} processors running AIX^{TM}.[1] In order to increase the utility of the open-source release, we decided to port the software to run on Linux© on the Intel 32-bit architecture (IA32), our second platform.

The porting activities focused on two milestones:

Functional port: establish a working version of the virtual machine on the second architecture as soon as possible, and

Performance port: achieve acceptable performance by the end of 2001.

This paper describes our experiences in working towards and reaching these milestones. The next section provides background on the Jikes RVM and an overview of the porting effort. Section 3 describes issues involved in getting the virtual machine up and running on a second platform. Section 4 presents the changes and innovations required to achieve good performance on that platform. Finally, section 5 concludes.

2 Overview

The Jikes RVM began life as the *Jalapeño virtual machine* in late 1997. The project had two design goals: 1) support high-performance Java^{TM} servers running on PowerPC multiprocessors under the AIX operating system, and 2) provide a flexible research platform "where novel virtual machine ideas can be explored, tested, and evaluated". Although it was written in the Java programming language, in the initial implementation portability was "*not* a design goal: where an obvious performance advantage can be achieved by exploiting the peculiarities of Jalapeño's target architecture ... we feel obliged to take it" [2].

Jikes RVM does not interpret bytecodes; rather it compiles each method to machine code and executes the machine code natively. In an adaptive Jikes RVM configuration, the *baseline* compiler performs the initial compilation of a method. Methods that are either frequently executed or computationally intensive are identified via a sampling mechanism and recompiled by the *optimizing* compiler [4].

The baseline compiler directly mimics the stack machine behavior of the JVM specification. The baseline compiler translates bytecodes to machine code quickly, but the resultant machine code typically runs slowly. The baseline compiler implementation de-

[1] There was also a partial port available for Linux/PowerPC, provided by collaborators from the University of Massachusetts.

pends heavily on the target instruction set architecture. Much of the work of a functional port lies in constructing a new baseline code generator, more or less from scratch.

The optimizing compiler expends more effort to produce high quality machine code for selected methods. The optimizing compiler implementation far exceeds the baseline compiler in size and complexity. However, most of the optimizing compiler does not depend on the instruction set architecture, reducing porting effort for a second architecture.

The Jikes RVM does not directly map Java threads of an application to operating system threads (POSIX pthreads). Instead, the system creates a virtual processor object for each pthread in use (normally one for each physical CPU). The Jikes RVM thread scheduler multiplexes the application's Java threads and the RVM's daemon threads onto these virtual processors.

Although the Jikes RVM is written in the Java programming language, it must perform actions (e.g. access registers and manipulate raw memory addresses) which cannot be expressed in the Java programming language [8]. The virtual machine provides a `VM_Magic` class to circumvent these restrictions [3]. The compilers do not translate the bytecodes of `VM_Magic` methods. Rather, the compilers recognize calls to these methods and inline custom machine code in place of the call.

The original implementation exploits the large PowerPC register set, with 32 general-purpose registers and 32 floating-point registers.[2] Adjusting to the register-scarce IA32 architecture presented a major challenge for this port. Differences in the instruction sets led to different calling and stack conventions for the two architectures. Writing in the Java programming language (explicitly big endian) shielded us from many, but not all endian problems[3] (For instance, with pre-loaded constants and when atomically updating bit and byte maps). Still, despite our initial indifference to the possibility of an eventual port, large portions of the Jikes RVM more or less worked on Linux/IA32 without modification.

As of February 1, 2002, the source code for the Jikes RVM itself contains approximately 203,000 lines of Java code, 18,000 lines of "meta" source files that are the inputs to several code generation tools, 6,000 lines of C++ code to interface with the operating system and to get the RVM started, and about 50 lines of assembly code to effectuate the initial transition from C++ to Java.[4] Most of this source code is independent of the target platform. The Java source files contain 162,000 lines of platform-independent code, 22,000 lines of PowerPC-specific code, and 19,000 lines of IA32-specific code. Approximately 6,000 lines of the "meta" source files are platform-independent, 3,600 are PowerPC-specific, and 8,400 are IA32-specific. The optimizing compiler comprises the largest subsystem of Jikes RVM, with 100,000 lines of Java source code; 78,100 are platform-independent, 14,200 lines are PowerPC-specific and 7,700 are IA32-specific. About 900 lines of the C++ operating system interface code are IA32-specific; 1200 are PowerPC-specific (these support both Linux and AIX). The assembly code is completely architecture-dependent. The *FullAdaptiveSemispace* configuration on Linux/IA32 represents a typical RVM build: it contains 821 Java classes comprising 225,000 lines of code (66,000 are machine generated).

3 Establishing functionality

A central aim of the functional port was to achieve a clean decomposition of the RVM into architecture-independent and architecture-specific pieces. The RVM was designed to be a *family* of virtual machines with different variants incorporating different memory management and compilation schemes as well as more minor peculiarities. It uses three mechanisms to achieve this variety: `static final` variables called *controls*, subdirectories of the file system, and a minimal preprocessor that allows limited conditional exclusion

[2] These registers are treated as dedicated, scratch, volatile (caller-save), or non-volatile (callee-save). Parameters are passed in volatile registers. Intermediate results can be accumulated in volatile or scratch registers. Non-volatile registers retain their value across method calls.

[3] PowerPC is "big endian" — the high-order byte of a word is at the lowest address within it — while IA32 is "little endian".

[4] In addition, the Jikes RVM source tree contains several tools used in the build process (5,600 lines) and the jdp debugger (33,300 lines).

of blocks of source text.[5] The port uses these mechanisms to support variants that run on Linux/IA32.

The remainder of this section discusses the specific issues that needed to be addressed to construct such variants.

3.1 IA32 Assembler

An IA32 assembler must contain a large body of complex and tedious code, full of odd special cases and strange idioms, simply because it must reflect the nature of the IA32 instruction set. For the IA32 port, this yields two obvious consequences: the two compilers should share a single assembler backend,[6] and this single backend should, in so far as is practical, be generated from specifications of the ISA to simplify attaining complete and correct coverage.

To accommodate the vastly different structures of the baseline and optimizing compilers, the shared assembler consists of two parts. The first consists of low-level code that generates binary code for specific IA32 instructions and operands; for example, it provides a function to emit a 32-bit add of a register and an immediate. It also has low-level support for other code generation miscellany, e.g. convenient support to generate forward branches. This low-level interface naturally suits the baseline compiler, which invokes it directly. The second part of the assembler processes the optimizing compiler's MIR (machine-dependent intermediate representation) instructions, and calls the appropriate low-level assembler routine for each one. This process involves examining the opcode and each operand of an MIR instruction, and, from them, determining which low-level assembler primitive to call.

Both levels of the assembler would be tedious and error-prone to write by hand. For the low-level functionality, we introduced a semi-automated approach. We first divided the IA32 instructions into equivalence classes based on instructions having similar legal sets of operands and similar generated bit patterns; for example, binary ALU operations such as ADD, SUB, AND, and XOR all fit similar formats. We then wrote a template for the low-level functions to generate each such equivalence class, and instantiated the template for each instruction in the class. This saved effort, and facilitated debugging, since an error in a template would occur in all its instructions and thus be more likely to show up in tests.

The higher-level assembler for the optimizing compiler is, if anything, even more complex; it consists of nested case statements depending on the operator of each instruction, and on properties of each of its operands. A stand-alone program generates this code fully automatically at build time, as follows. Text files holding tables define the MIR instruction formats. For each MIR operator, the program examines the low-level assembler for functions generating that opcode, and generates a table of the operand types those functions support. It then generates a tree of queries of the MIR instruction operands to determine which low-level function to call. The generator also inserts error-checking code to catch any instruction that cannot be assembled.

3.2 Baseline Compiler

A baseline compiler actually consists of two main components of roughly equal size: a target-independent portion that is responsible for generating GC maps and other descriptive information and a target-specific code generator. The heart of baseline compiler code generation executes a switch statement that emits code for each Java bytecode. Most of these 209 cases present simple exercises that are "solved" by emitting a dozen or fewer straight line assembler instructions. For those cases that are more complicated — for instance, the seven bytecodes that sometimes entail class loading — the internal structure of the case was imported from the PowerPC baseline compiler. We were careful to verify that each case (except for some of the "wide" variants) were exercised against a library of bytecode tests developed in conjunction with that original baseline compiler.

[5] To keep down instances of this rather ugly feature, we try to limit it usage to three cases: to define a control, to reference a class that would not otherwise be loaded, or to define or reference a field that would not otherwise be needed. In the 162,000 lines of platform independent Java code, there are 38 preprocessor blocks impacting 479 lines of source code that are related to the choice between AIX/PowerPC or Linux/IA32.

[6] The original PowerPC compilers each had their own assemblers.

3.3 Optimizing Compiler

Because writing an optimizing compiler represents a significant investment, from the beginning we designed the optimizing compiler for portability. A key aspect of this design divides the intermediate representation (IR) into three levels of operators:

1. **High Level IR (HIR)**: The HIR operator set is architecture independent and resembles the bytecode instruction set, although the IR uses a register transfer language in place of the bytecode stack abstraction.
2. **Low Level IR (LIR)**: The LIR operator set is architecture independent and resembles the instruction set of a typical RISC machine. The main difference between HIR and LIR is that complex HIR operators such as `new` or `call virtual` are expanded into the appropriate sequence of primitive operations.
3. **Machine Level IR (MIR)**: The MIR operator set is architecture-specific; with the exception of a few pseudo-operators that are expanded as part of final assembly, the MIR provides a one-to-one mapping between operators and the target ISA.

Translation from bytecodes to HIR, HIR optimizations, translation from HIR to LIR, and LIR optimizations are all architecture independent.

At system build-time, the builder generates classes representing the IR operators and helper functions from template files. Thus, the task of defining MIR operators corresponding to IA32 instructions consisted of defining new instruction formats and operator characteristics in two machine-dependent template files. The MIR operator definitions for IA32 consist of about 1300 lines in two files, defining 153 operators falling into 29 formats.

After defining the MIR, the main tasks in porting the optimizing compiler to a new ISA involve implementing the three major architecture-specific compiler phases: translation from LIR to MIR (aka instruction selection), register allocation, and final assembly. Each of these stages actually contains both architecture-independent and architecture-specific portions; we adopted a common idiom to define abstract classes that implement the shared functionality in terms of abstract methods defined by architecture-specific subclasses.

The instruction selection phase translates the machine-independent LIR into architecture-specific MIR. This translation phase partitions the dependence graph of each basic block into a forest of trees, and feeds the forest to a Bottom-Up Rewrite System (BURS)-based tree-pattern matching system [7]. Thus, the major porting task is to define the tree patterns and associated actions that serve as the input grammar to the BURS engine.

The register allocator maps the infinite set of symbolic registers onto a finite set of physical registers and spill locations. In addition, this phase generates prologues, epilogues, and calling sequences that respect the Jikes RVM and/or native OS calling conventions. The optimizing compiler relies on a variant of linear-scan [9] register allocation. The core of the register allocator should be machine-independent, but the implementation handles a fair number of low-level architecture-specific issues. Unfortunately, the original PowerPC implementation deeply intertwined the machine-dependent and machine-independent register allocator code. Rather than tackle a major refactoring problem with the old implementation, we decided to re-implement the register allocator from scratch for IA32. The new implementation cleanly separates machine-dependent and machine-independent code and includes expanded functionality and heuristics to suit both register-scarce and register-rich architectures. We have since back-ported the new implementation to PowerPC, so both platforms now share the machine-independent portion of the register allocator.

Final assembly generates executable machine code from the MIR and finalizes descriptive data structures such as exception tables and GC maps. The code for generating the descriptive data structures does not depend on the target architecture. Furthermore, as described above, the build system mechanically generates code that interfaces the MIR to the assembler. Therefore this stage introduced little work for porting.

The optimizing compiler also performs some optimizations on the MIR. The main MIR peephole optimizations, branch simplification and null check folding, do not depend on the architecture and worked immediately on IA32. We have not yet ported the instruction scheduler to IA32.

One difficulty in generating IA32 code compared to PowerPC is dealing with register restrictions imposed by the non-orthogonal instruction set architecture. Figure 1 shows some examples.

For some IA32 instructions, a particular operand *must* reside in a distinguished register. For example, in Figure 1b, the value of `t2` must reside in `ecx` at the shift instruction (`shl_acc`). We represent this information in the IR by explicitly assigning `t2` to `ecx` before the shift instruction during instruction selection for the shift (Figure 1c). The register allocator respects liveness for physical registers, and will further attempt to allocate `t2` to `ecx` to remove the copy operation. Similarly, in our calling convention `eax` holds the return value; we enforces this by inserting a copy from the symbolic return value register to `eax`, as shown in Figure 1c.

When BURS inserts a memory operand, the register allocator must respect further restrictions. For example, if the allocator were to spill `t0` in Figure 1c, this would force a later pass to move `t0` to a scratch register before its use in the `movsx` memory operand. For this reason, the linear scan spill heuristics consider a spill of a symbolic register used in a memory operand to be more expensive than a spill of a register operand that could be replaced by a memory operand representing the spill location.

Furthermore, for many instructions, the IA32 architecture dictates that only four of the eight general-purpose registers can hold 8-bit values. For example, in Figure 1c, `t3` can reside in the low word of `eax, ebx, ecx`, or `edx`; but not, for example in `ebp` or `edi`. The register allocator handles these types of restrictions with special case code, computing the restrictions as a pre-pass to register allocation.

3.4 Other VM Subsystems

The other VM subsystems — memory management, thread scheduling, locking, class loading, dynamic type checking, etc. — ported without change to Linux/IA32. Our use of Java as an implementation language shielded this code (some of it very low level) from any target dependencies. The only exceptions occur in about half a dozen sequences which use `VM_Magic` methods to perform direct loads or stores of byte quantities; the address computation needed to be parameterized on whether the target platform was big or little endian.

3.5 VM Conventions

Details of IA32's `CALL` and `RET` instructions forced major differences in stack and calling conventions. `CALL` pushes the return address on the stack and then branches to an indicated address. `RET` pops a return address off the stack and branches to it (discarding an indicated number of parameter bytes in the process).

On AIX the return address is saved at a fixed address in the *caller*'s stackframe. Using `CALL` effectively prevents this since the relative address of the stacktop off the frame pointer varies from call-site to call-site. To be conveniently accessible at all, the return address must be at a fixed address in the *callee*'s stackframe. Thus, on IA32 the return address starts a new stackframe.

This introduces a number of complications some of them minor.

First, the header is at the bottom of an AIX stackframe but the top of an IA32 stackframe (stacks growing down from high memory in both cases). This does not present a stack addressing problem: stack offsets are positive on AIX, negative on IA32.

Second, the ordering of fields in the headers of stackframes differs on the two architectures. This did not cause a problem since the header fields are always accessed with `static final` constants off the frame pointer. These constants differ on the two architectures.

Third, the size of stackframes, which is fixed (per method) and known *a priori* on AIX, varies from call-site to call-site on IA32. On

```
byte foo(byte[] a,
int x, int y){
 int i=x<<y;
 return a[i];
}
```

a)

```
t1 = shl_acc t2
t3 = byte_aload t0,t1
return t3
```

b)

```
ecx = t2
t1 = shl_acc ecx
t3 = movsx [t0+t1]
eax = t3
return eax
```

c)

```
ecx = [esp + 12]
edx = shl_acc ecx
eax = movsx [eax+edx]
return eax
```

d)

Figure 1: Examples of architectural register restrictions in the optimizing compiler IR. a) Java source code; b) IR fragment before instruction selection; c) IR fragment before register allocation; d) IR fragment after register allocation.

AIX it is natural to check for stack overflow when allocating a new stackframe; on IA32 this requires explicitly testing against a calculated upperbound of the eventual size of the stackframe.

Finally, stack-walking (e.g. during exception handling or garbage collection) was severely complicated by the fact that the return address was in the caller stackframe on AIX and the callee stackframe on IA32. The code difference was finally minimized by adopting the convention that the return address would be computed immediately before moving from callee to caller. On AIX this entails a redundant load off the contents of the callee's frame pointer. But, since stack-walking is not expected to be performance critical, we tolerate the pain in the interest of compatibility.

To facilitate the functional port, we initially added an extra word to the header of an IA32 stackframe. Whenever a method was called, the return address in the callee's stackframe header was copied into the new slot in the caller's header. This allowed immediate utilization of code that assumed the AIX convention. However, restructuring this common code so as to eliminate the need for this redundant header word was an ongoing porting headache for several months.

3.6 Synchronization

The PowerPC and IA32 architectures have different mechanisms for synchronizing multiprocessors.

The PowerPC architecture uses a weak memory consistency model. The PowerPC instruction set includes two synchronization instructions: `lwarx` and `stwcx`. The first of these loads a word from an address memory while setting a reservation for the executing processor on this address (any reservation another processor may have on the address is cleared as a side effect). The second stores a word at an address provide the executing processor holds a reservation on the address. The success or failure of this operation is recorded in a condition register.

The IA32 architecture enforces stronger memory consistency among the multiple processors. The IA32 instruction set has a compare-and-swap instruction (`cmpxchg`): the value at a specified address is compared to a second value, if the two are equal, a third value is stored at the address. The success or failure of the operations can be obtained from a machine register. A locking prefix byte to this instruction makes its behavior appear atomic to any other processors.

The initial PowerPC implementation of Jikes RVM used methods of the `VM_Magic` class to directly emit `lwarx` and `stwcx` instructions. Rather than create architecture-specific classes for all the methods that called these methods, we designed `VM_Magic` methods that could be used on either architecture but whose implementations were architecture specific. We developed a synchronization idiom whereby attempting to perform a synchronized write first requires obtaining the old value using a *prepare* operation and then issuing an *attempt* operation which takes both the old and a new value as well as the raw address. Higher level pseudo-primitives, such as `fetch-and-add`, were implemented in Java using this discipline and provided as runtime utilities.

The `int VM_Magic.prepare()` method takes a raw address as its only parameter. On the IA32 architecture this is implemented as an ordinary load instruction. On the PowerPC it is a lwarx instruction.

The boolean VM_Magic.attempt() method takes a raw address and two (32 bit) values as parameters. On the IA32, it causes the corresponding atomic compare-and-swap to be executed. On the PowerPC, the second parameter is ignored, while the other two are used by a stwcx instruction.[7] In either case, the success of the operation is returned as the result of the method.

3.7 Operating system issues

The thin layer of C/C++ code that interfaces between the RVM and the operating system was ported from AIX to Linux either without change or by replacing the invocation of a system function on AIX with its Linux equivalent. We rewrote for IA32 less than half a page of assembly code which transfers initial execution to the RVM image. The only system service which proved troublesome was the Linux POSIX thread (pthread) library where we had troubles both with the earlier implementation of the library and with differences between the AIX and later Linux implementations.

We started this work on the 2.2 Linux kernel and associated libraries. For this Linux release, the pthread library computes the identity of a thread as a function of the ESP (stack pointer) machine register. Since the RVM virtual processors multiplex several Java threads, each with its own stack (none of them beginning on the large power of two boundary expected by the library), and since our implementation used the ESP register to address the stack of the running thread, we cannot run with this pthread library. (On 2.2 Linux, the RVM only runs with a single virtual processor.)

Fortunately, the 2.4 Linux/IA32 release (and some earlier development releases) resolved this problem.[8] We also found some differences in the behavior of POSIX threads on AIX and Linux. On AIX, one process may have many pthreads. On Linux, each pthread looks and acts like a separate process. One area where this difference manifests is the behavior of the system with respect to signal handling. The RVM uses two signal handlers which are not reentrant and cannot execute simultaneously. On AIX, we specified that these signals were to be masked for the duration of the signal handler, and as expected pending signals wait until earlier executions of a signal handler finish before executing. Linux did not follow this behavior, so the Linux signal handlers need to provide their own explicit synchronization.

4 Improving performance

The functional port of Jikes RVM to Linux/IA32 was mostly complete by the end of August 2001. The initial port achieved 35% of our performance target. Over the course of the next five months, Jikes RVM Linux/IA32 performance more than doubled to reach 95% of the IBM 1.3.0 DK. Figure 2 shows how the performance[9] increased over this time period, and Figure 3 shows relative performance for each of the SPECjvm98 benchmarks[10] as of February 2002. The results show that Jikes RVM performance is competitive overall, but lags behind the IBM DK on the two floating-point codes (mpegaudio and mtrt) and on compress (an array-based set of tight nested

[7] On the PowerPC architecture, the baseline compiler expends unnecessary extra work pushing the unused middle parameter onto the stack. However, the optimizing compiler identifies the computation as dead and eliminates it. Thus, the optimizing compiler produces code for the prepare and attempt primitives that is as efficient as architecture-specific primitives.

[8] It remains in the 2.4 Linux/PowerPC release.

[9] The Jikes RVM FastAdaptiveSemispace images, used for Figures 2 and 3, gives the best overall performance due to feedback-directed optimization and delayed compilation effects [4]. However, to reduce the impact of timer-driven non-deterministic actions, the remainder of this paper reports results using a (non adaptive) FastSemispace image which compiles each method the first time it is invoked at optimization level O2. In all cases, two virtual processors and a 400 MB heap were used. All runs are on a 4-way IBM Netfinity with 700MHz Pentium^{TM} III processors and 3 GB of memory. The Linux installation is a customized RedHat version with a 2.4.12 kernel and GNU Libc version 2.2.4; the libc and kernel versions support the version of LinuxThreads that uses the GS segment register for thread-local storage enabling it to support Jikes RVM's user-level multithreading mechanism.

[10] These benchmarks were developed by the Standard Performance Evaluation Corporation [6]. The performance numbers reported in this paper are the best run of 5 on each individual SPECjvm98 benchmark. These runs do *not* conform to the official SPEC run rules, so our results do not directly or indirectly represent a SPECjvm98 metric, and are not comparable with a SPECjvm98 metric.

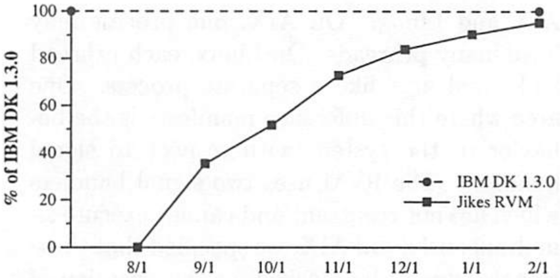


Figure 2: Monthly performance of Jikes RVM on the SPECjvm98 benchmarks as a percentage of the performance of the IBM 1.3.0 DK for Linux/IA32 from August 1, 2001 through February 1, 2002.

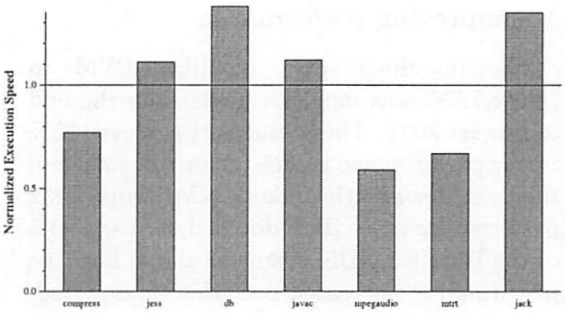


Figure 3: Performance of Jikes RVM on the individual SPECjvm98 benchmarks as a percentage of the performance of the IBM 1.3.0 DK for Linux/IA32 on February 1, 2002.

loops).

This section of the paper describes the main IA32 specific enhancements made to improve Linux/IA32 performance.[11] The first section discusses performance-motivated changes to the VM's register conventions. The next two sections describe enhancements to the optimizing compiler's instruction selection and register allocation phases. Generating even mediocre IA32 floating point code was challenging; section 4.4 describes some of the alternatives we explored.

4.1 VM Register Conventions

The initial functional port followed the PowerPC implementation in dedicating four (of the eight IA32 "general purpose") registers: a pointer to the currently executing stack-frame (FP), a pointer to a region of static data (JTOC), an indirect pointer to thread-local storage for the current Java thread (TI), and a pointer to pthread-local storage associated with the current virtual processor (PR). In addition, it dedicated `esp` as a stack pointer, leaving only three registers for general use. It soon became apparent that good performance would hinge in part on freeing up some of these dedicated registers.

We first reclaimed the TI and JTOC registers. Instead of dedicating registers to hold these values, the system now caches these values in the pthread-local storage accessed by the PR register. This strategy adds an extra indirection to access Java thread-local and static storage.

Next, we reclaimed the frame pointer register. This change required more intrusive system modifications. As with TI and JTOC, the system caches the current frame pointer register in pthread-local storage. Each compiler was modified to maintain this frame pointer field in the prologue and epilogue sequences. The C trap handler that handles a hardware trap or software interrupt was modified to acquire the frame pointer indirectly through the PR register instead of from a register. Additionally, each compiler was modified to manage stack storage solely off the stack pointer (`esp`), with no reliance on the frame pointer. There were also some complications with IA32 baseline compiler assumptions for low-level details of stack resizing and GC map computation, which fall beyond the scope of this paper.

The current system has two dedicated IA32 registers: SP (`esp`) and PR (`esi`). The remaining six GPR registers are available for register allocation. We have considered reclaiming PR as a non-volatile by using an IA32 segment register as a pointer to pthread-local storage. Three considerations have so far deterred us from making the attempt. First, since the segment registers can't address arbitrary words in memory, it would be difficult to encode the structures that they point to a ordinary Java objects. Second, we do not feel

[11] During the six month period shown in Figure 2 several new platform independent optimizations were added to the optimizing compiler and existing optimizations were enhanced. Although these contributed to the performance improvements shown in the graph, most of the gain was caused by IA32 specific improvements (during the same period AIX/PowerPC performance only improved by about 10%).

certain that Linux does not, or, more importantly, will not in the future use any particular segment register for its own purposes. Third, we are concerned that on some IA32 implementations segment register access might be prohibitively slow.

4.2 Instruction Selection

The heart of the instruction selection phase relies on a BURS-based tree-pattern-matching system. We have extended iburg [7] to generate Java code (instead of C) and work on a general dependence graph. The key idea is to partition the dependence graphs into a forest of expression trees based on their register-true dependencies. The BURS pattern matching system then processes each tree in the forest to perform instruction selection and emit code, one tree at a time, in an order that respects the inter-tree ordering constraints encoded by the original dependency graph [5, 10].

In the initial port, we defined a "bare bones" grammar that described a straightforward translation of the 124 LIR operators into MIR operators using 142 rules and 7 non-terminals. As the performance work progressed, the grammar tripled in size. The current grammar includes additional patterns for optimizing floating point computations and conditional branching, exploiting memory operands, and other miscellaneous enhancements such as recognizing complex addressing modes, exploiting special instructions such as LEA, TEST, INC, etc., and avoiding needless sign extension of byte/short loads. Table 1 reports the contribution of each group of enhancements to the size of the rules.

In addition to extending the grammar as described above, we also enhanced our BURS driver with heuristics to reduce register pressure. We label each tree node with an estimate of the number of live values required to compute it, using the algorithm from section 9.10 of the Dragon book [1]. The BURS driver uses this numbering to choose which child node to emit first when generating code for a given tree and to select from the set of ready trees[12] which tree to emit next. In both cases, choosing the node/tree with the largest number of registers first tends to reduce the number of simultaneously live values and thus reduce register pressure. As reported in section 4.3, this heuristic made a small but measurable difference in overall performance and was extremely simple to implement.

Benchmark	% Speedup
compress	1.9
jess	7.2
db	1.4
javac	-0.5
mpegaudio	52.3
mtrt	5.1
jack	8.0
geo. mean	9.7

Table 2: Percentage speedup obtained by Complete rules over the initial Basic rules.

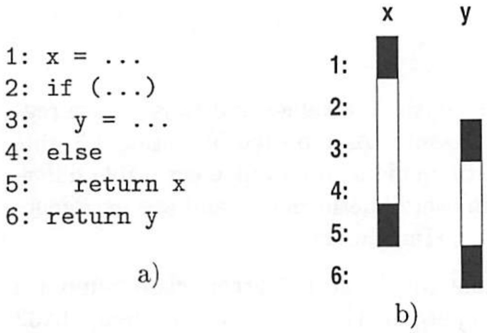
While the PowerPC architecture supports three-operand ALU operations (a=b+c), IA32 supports two-operand ALU operations(x+=y). The IA32 compiler converts the three-operand LIR to two-operand form in a pre-pass to BURS, which tripped some subtle issues. Initially, this pre-pass took obvious actions to avoid introducing useless move instructions. For example, it would transform a=a+b to a+=b. In other cases, it would use local liveness information to avoid inserting moves. For example, if b is dead after a=b+c and this is the only definition of a, then the pass would emit b+=c and the replace uses of a with uses of b. This second optimization tripped a subtle difficulty; it can hinder instruction selection of other expression trees within the basic block by introducing additional inter-tree anti and output dependencies and by extending the live range of b beyond the current basic block. Thus, in some cases it is actually better to translate a=b+c into b+=c; a=b and rely on the register allocator to coalesce away the move instruction.

Table 2 shows the performance improvements obtained by adding all of the enhancements to the basic grammar. The most important enhancement was adding rules to exploit the floating point stack (mpegaudio, mtrt), but the other enhancements also resulted in modest gains. The impact of instruction selection and register allocation on floating point performance is explored in more detail below.

[12] A tree is ready if all of the trees on which it is dependent have already been emitted.

Description	Number of Rules	Number of Non-terminals
Basic	142	7
Floating Point Stack	90	2
Conditional Branches	42	6
Memory Operands	170	8
Misc. Other Enhancements	144	2
Complete	588	25

Table 1: The Basic grammar was enhanced with 446 rules and 18 non-terminals to support optimizations in instruction selection. BURS must recognize 124 LIR operators.



```
1: x = ...
2: if (...)
3:   y = ...
4: else
5:   return x
6: return y
```

Figure 4: Example of linear scan live intervals with holes.

4.3 Register Allocation

The optimizing compiler relies on a variant of linear-scan [9] register allocation. To give the register allocator more freedom, we implemented a variant of linear scan that deals with holes in live ranges. Consider Figure 4. The basic linear-scan algorithm would not allocate x and y to the same physical register, as their live intervals (denoted by the rectangular bars) overlap. However, our enhanced algorithm represents the live intervals with holes, as represented by the black-shaded areas in the Figure. As a result, our allocator could allocate x and y to the same physical register.

Traub et al. [11] previously described a linear scan variant to deal with holes. Our approach differs in two ways.[13] First, we perform the intersection of two sparse live intervals, rather than insisting on perfect nesting of one interval within another. Although this introduces super-linear complexity to the algorithm, we do not believe this causes major problems in practice.

Secondly, Traub et al.'s algorithm splits live ranges on the fly, with a post-pass clean-up phase to reconcile differences. In contrast, our algorithm marks certain symbolic registers as spilled. A post-pass clean-up phase deals with the spills. If an instruction uses a spilled register, the clean-up phase either introduces a memory operand referring to the spilled memory location, or moves the spilled value into a scratch register. On PowerPC, the original register allocator reserved three registers for use as scratch, so finding a free scratch register was almost always trivial. On IA32, we did not reserve any scratch registers, so the register allocator must create scratch registers upon demand.

Figure 5 details the algorithm for dealing with spilled values. The algorithm makes one pass over the statements. As it progresses, the algorithm keeps track of which symbolic register values are cached in each scratch register. Before each statement, the algorithm vacates any scratch registers which are needed for the next statement. Then, the algorithm processes a statement. It attempts to replace spilled symbolic registers with memory operands representing the spill location. If this is infeasible, the algorithm chooses a physical register as a victim to be vacated, so the victim can be used as a scratch register. The victim will continue to cache the symbolic value until either the physical register is needed for a future statement, or the victim is chosen to cache a different symbolic register. Scratch register mappings are not maintained across basic block boundaries; all victims are vacated at block exits.

As a side effect of vacating and introducing scratch registers, the code updates stack maps required for type-exact GC.

[13] We have not performed an apples-to-apples comparison of our algorithm compared to Traub et al.

```
for each statement s do
  if s can leave the basic block via a call, jump, fall-through, or exception then
    vacate cached value and restore original value for each scratch register before s
  vacate cached value and restore original value for any scratch register used by s
  for each spilled symbolic register r in s do
    if r is currently cached in scratch register p then
      replace r with p in statement s
    else if s needs a scratch register for r then
      choose a scratch register victim p to hold r in s
      vacate current value of p
      cache value of r in p
      replace r with p in statement s
    else replace r with a memory operand representing r's spill location
  done
done
```

Figure 5: Algorithm for dealing with spilled values after linear scan.

With the IA32 limited register set, spills are common and have a huge impact on performance. Subsequent to the initial functional port, we introduced several heuristics and optimizations to improve performance.

We now evaluate the following heuristics to reduce register pressure. We provide only a high-level overview of each heuristic; consult the open-source code for more details.

Intelligent scratch victim selection The initial implementation of the algorithm in Figure 5 chooses a victim arbitrarily. Furthermore, the initial implementation does not use liveness to determine whether the victim needs to be vacated and restored. This heuristic uses liveness computed during linear scan and attempts to choose a victim that does not currently hold a live value. Furthermore, this optimization uses the same information to avoid vacating and restoring dead values.

Smarter linear scan spill heuristic When facing a spill situation, the original linear scan implementation chooses a symbolic register to spill at random. This heuristic estimates the cost of spilling based on appearances of the register, weighted by loop depth, and chooses spill candidates accordingly.

Register-pressure-aware BURS This optimization introduces a heuristic into instruction selection to attempt to generate code in an order to minimize overlapping live ranges. When faced with multiple expression trees which can be emitted in any order, this heuristic chooses the tree that consumes the most register values.

Global Coalescing We implemented a separate pass to coalesce registers; basically, if there is a `MOV x = y`, if the live ranges of x and y do not overlap, then we replace all appearances of y with x.

Figure 6 shows the marginal performance improvement gained by enabling each of these four optimizations cumulatively, compared to performance with none enabled. Thus, the bar labeled "Scratch Victim" enables intelligent scratch victim selection; the next bar "LS Spill" further enables the linear scan spill heuristic, etc.

Altogether, the register pressure heuristics improve performance on average by 15%. The results show that the spill heuristics and scratch register selection heuristics are most effective across the board. Anomalously, the scratch victim heuristic hurts `mpegaudio`; we currently have no explanation for this. Instruction selection re-ordering has a minimal impact, slightly improving `compress` and `db`. Coalescing is usually insignificant; except it causes a substantial improvement for `mpegaudio`. `Mtrt` degrades as we add more optimizations; however, the floating-point results here should be taken with a grain of salt, since

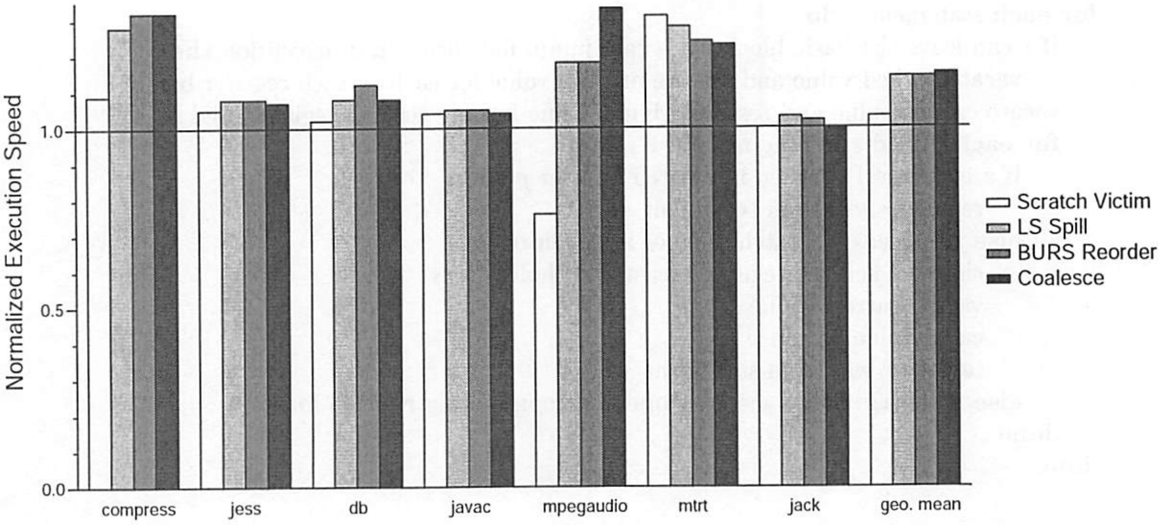


Figure 6: Relative performance gains by enabling, cumulatively, four optimizations designed to reduce register pressure.

Figure 3 shows that Jikes RVM IA32 floating-point performance is mediocre, even with all optimizations enabled.

4.4 Floating point

The IA32 architecture provides an abstraction of a floating-point stack, a sharp difference from the flat floating-point register set of the PowerPC.

In the original functional port, in order to minimize changes to the system, we treated the floating-point stack locations as independent physical floating-point registers. During instruction selection, BURS treated symbolic floating-point registers just like symbolic integer registers. The linear scan register allocator allocated the symbolic floating-point registers to seven floating-point stack locations, as if these were seven physical registers. The eighth stack location was reserved for use as a scratch register downstream, in order to generate code that moves values between stack locations and to memory.

This original scheme had the advantage that the linear scan allocator had full freedom to allocate the stack locations using global analysis. However, this scheme has a severe drawback. Since the BURS instruction selection saw only orthogonal symbolic floating-point registers, it could not generate code to exploit the stack operations available in the IA32 instruction set.

An alternative scheme could allow BURS to generate floating-point stack code freely within a basic block. With this scheme, instruction selection could use the floating-point stack resources freely within a basic block. However, since the linear scan algorithm does not understand stack locations, it could not allocate values to stack locations across basic blocks. In effect, all register allocation would be constrained to a single basic block, spilling values to memory across basic blocks.

We chose a hybrid scheme. We give instruction selection the freedom to place a floating-point value either on the floating-point stack or in a symbolic floating-point register. The register allocator allocates symbolic registers to free stack locations. Note that if BURS allocates a value to a floating-point stack location, that stack location is not available for use by the register allocator. We model this by inserting dummy def and use instructions for physical stack locations reserved by instruction selection.

Table 3 compares performance on the two floating-point SPECjvm98 codes. The Table shows that each technique helps `mpegaudio`, but shows an anomaly where inter-block register allocation hurts `mtrt`. Our initial functional port used only the "RA" register allocation strategy, as this option most closely matches the extant PowerPC port. Later we also added the BURS floating-point stack code

	None	RA only	BURS only	Both
mpegaudio	1	1.548	1.544	1.957
mtrt	1	0.668	1.251	1.181

Table 3: Performance comparison of alternative floating-point code generation strategies (speed normalized to "None"). "RA" allows inter-block register allocation, while "BURS" allows intra-block generation of floating-point stack code for expressions.

generation. We didn't seriously consider the other two possibilities, but include them to enable comparisons.

Although we have improved RVM floating point performance compared to the initial functional port, performance still lags behind the IBM product DK. We still face the two anomalies reported for floating-point performance: recall that the smart scratch victim selection hurts `mpegaudio` and floating point register allocation degrades `mtrt`. We have not yet investigated these anomalies, and we hope to improve Jikes RVM floating-point performance in the future.

5 Conclusions

We have described our experiences porting a high-performance virtual machine to its second architecture. In the process, we have endeavored to enforce a clean separation between architecture-dependent and architecture-independent code. As a result, we expect that a port to a third 32-bit architecture would be much easier.

A major issue only partially addressed is migration to a 64-bit architecture. Recently, a `VM_Address` type has been introduced to statically isolate code that manipulates raw addresses in the VM implementation (originally Jikes RVM used the type `int` to represent raw addresses, making it impossible to statically isolate such code). However, work still remains to find and update all code in the VM implementation that assumes that reference/address values are four byte quantities.

Substantial work remains to be done in the optimizing compiler. Clearly, opportunities remain to improve performance on both PowerPC and IA32. On IA32 in particular, floating-point performance still lags behind the IBM DK 1.3.0. We have recently implemented live range splitting to help further reduce register pressure, but have not yet seen substantial performance improvements. Further optimization passes may also help; one optimization not yet enabled is instruction scheduling, which could help reduce register pressure and/or increase instruction-level parallelism. Also, it may be an interesting research topic to determine how to constrain HIR optimizations, such as SSA conversion and redundancy elimination, to reduce register pressure.

Since the open-source release in October 2001, we are aware of several efforts by academic and other researchers to help address some of these concerns. We understand that significant progress has been made porting to IA32 on Win32, and to 64-bit PowerPC. We hope these and other enhancements will make their way into the open-source code base, so that Jikes RVM will further mature as a platform for programming language implementation research.

Acknowledgments

We thank all the members of the Jalapeño team who have contributed to the code base over the past four years. We especially thank Vivek Sarkar and Mauricio Serrano, who contributed key ideas in the portable design of the optimizing compiler. (Mauricio also warned us that we would get in trouble in a port to IA32 if we associated the return address with the caller's stackframe. He was right.) We additionally extend special thanks to Mike Hind and Mark Mergen for their efforts in support of this work. Also thanks to Dave Hansen for making the `iburg` tool available, and to Steve Blackburn for his help and feedback.

Jikes RVM is available under the Common Public License (CPL) from `www.ibm.com/developerworks/oss/jikesrvm`.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[2] B. Alpern, D. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. Fink, D. Grove, M. Hind, S. Flynn Hummel, D. Lieber, V. Litvinov, T. Ngo, M. Mergen, J. R. Russell, V. Sarkar, M. J. Serrano, J. Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalepeno virtual machine. *IBM Systems Journal special issue on Java performance*, 39(1), 2000. (see also http://www.research.ibm.com/jalapeno).

[3] B. Alpern, D. Attanasio, J. J. Barton, A. Cocchi, S. Flynn Hummel, D. Lieber, T. Ngo, M. Mergen, J. Shepherd, and S. Smith. Implementation of Jalepeño in Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, November 1999.

[4] M. Arnold, D. Grove, S. Fink, M. Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the ACM SIGPLAN* Conference on Object-Oriented Programming Systems, Languages, and Applications *(OOPSLA 2000)*, Minneapolis, MN, Oct. 2000. Also published as ACM SIGPLAN Notices, volume 35, number 10.

[5] M.G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M.J. Serrano, V.C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Dynamic Optimizing Compiler for Java. In *ACM Java Grande Conference*, June 1999.

[6] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98/, 1998.

[7] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering, a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.

[8] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* The Java Series. Addison-Wesley, 1996.

[9] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, September 1999.

[10] Vivek Sarkar, Mauricio Serrano, and Barbara Simons. Register-sensitive selection, duplication, and sequencing of instructions. In *ACM International Conference on Supercomputing*, June 2001.

[11] Omri Traub, Glenn Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. In *SIGPLAN '98 Conf. on Programming Language Design and Implementation*, pages 142–151, May 1998.

A Modular and Extensible JVM Infrastructure*

Patrick Doyle and Tarek S. Abdelrahman
Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario, Canada M5S 3G4
{doylep,tsa}@eecg.toronto.edu

Abstract

This paper describes the design, implementation, and experimental evaluation of a modular and extensible Java® Virtual Machine (JVM) infrastructure, called Jupiter. The infrastructure is intended to serve as a vehicle for our research on scalable JVM architectures for a 128-processor cluster of PC workstations, with support for shared memory in software. Jupiter is constructed, using a building block architecture, out of many modules with small, simple interfaces. This flexible structure, similar to UNIX® shells that build complex command pipelines out of discrete programs, allows the rapid prototyping of our research ideas by confining changes in JVM design to a small number of modules. In spite of this flexibility, Jupiter delivers good performance. Experimental evaluation of the current implementation of Jupiter using the SPECjvm98 benchmarks shows that it is on average 2.65 times faster than Kaffe and 2.20 slower than the Sun Microsystems JDK (interpreter versions only). By providing a flexible JVM infrastructure that delivers competitive performance, we believe we have developed a framework that supports further research into JVM scalability.

1 Introduction

The use of the Java® programming language has been steadily increasing over the past few years. In spite of its popularity, the use of Java remains limited in high-performance computing, mainly because of its execution model. Java programs are compiled into portable stack-based *bytecode* instructions, which are then interpreted by a run-time system referred to as the Java Virtual Machine (JVM). The limited ability of a Java compiler to optimize stack-based code and the overhead resulting from interpretation lead to poor performance of Java programs compared to their C or C++ counterparts.

Consequently, there has been considerable research aimed at improving the performance of Java programs. Examples include: just-in-time (JIT) compilation [1, 2], improved array and complex number support [3, 4], efficient garbage collection [5, 6], and efficient support for threads and synchronization [1].

The majority of this research has focused on improving performance on either uniprocessors or small-scale SMPs. Our long-term research addresses scalability issues of the JVM for large numbers of processors. In particular, our goal is to design and implement a JVM that scales well on our 128-processor cluster of PC workstations, interconnected by a Myrinet network, and with shared memory support in software. However, in order to carry out this research, we require a JVM infrastructure that allows us to rapidly explore design and implementation options. While there exist a number of JVM frameworks that we could use [1, 7, 8, 9], these frameworks provide limited extensibility and are hard to modify. Hence, we embarked on the design and implementation of a modular and extensible JVM, called Jupiter. It uses a building block architecture which enhances the ability of developers to modify or replace discrete parts of the system in order to experiment with new ideas. Further, to the extent feasible, Jupiter maintains a separation between orthogonal modifications, so that the contributions of independent researchers can be combined with a minimum of effort. In spite of this flexibility, Jupiter supports simple and efficient interfaces among modules, hence preserving performance. In this paper, we

*This work was supported by research grants from NSERC and CITO. Patrick Doyle was supported in part by a scholarship from Sun Microsystems.

focus on this Jupiter infrastructure. In particular, we describe the overall architecture, various implementation aspects, and performance evaluation of Jupiter.

The current implementation of Jupiter is a working JVM that provides the basic facilities required to execute Java programs. It has an interpreter with multithreading capabilities. It gives Java programs access to the Java standard class libraries via a customized version of the GNU Classpath library [10], and is capable of invoking native code through the Java Native Interface [11]. It provides memory allocation and collection using the Boehm garbage collector [12]. On the other hand, it currently has no bytecode verifier, no JIT compiler, and no support for class loaders written in Java, though the design allows for all these things to be added in a straightforward manner. The performance of Jupiter's interpreter makes it comparable to commercial and research interpreters, while still maintaining a high degree of flexibility.

The remainder of this paper is organized as follows. In Section 2 we give an overview of Jupiter's architecture. In Section 3 we present details of Jupiter's design and implementation. In Section 4 we present the results of our experimental evaluation of Jupiter. In Section 5 we give an overview of related work. Finally, in Section 6 we provide some concluding remarks.

2 System Architecture

The overall structure of Jupiter is depicted in Figure 1. In the center is the `ExecutionEngine`, the control center of the JVM, which decodes the Java program's instructions and determines how to manipulate resources to implement those instructions. The resources themselves are shown as ovals, and include Java classes, fields, methods, attributes, objects, monitors, threads, stacks and stack frames (not all of which are shown in the diagram).

The responsibility for managing each resource is delegated by the `ExecutionEngine` to a particular *Source* class, each shown as a rectangle within the pie slice that surrounds the resource it manages. The Sources insulate the `ExecutionEngine` from the details of how resources are managed. Sources share a simple, uniform interface: every Source class has one or more `get` methods which return an instance of the appropriate resource. Each `get` method has arguments specifying any information needed by the Source to choose or allocate that resource, and the Source is responsible for deciding how the resource should be created, reused, or recycled.

An incarnation of Jupiter, then, is constructed by assembling a number of Source objects in such a way as to achieve the desired JVM characteristics, a scheme referred to as a *building-block architecture* [13]. As each Source is instantiated, its constructor takes references to other Sources it needs in order to function. For instance, as shown in Figure 1, the `ObjectSource` makes use of a `MemorySource`, so the `ObjectSource` constructor would be passed a reference to the particular `MemorySource` object to which it should be connected. The particular Source objects chosen, and the manner in which they are interconnected, determines the behaviour of the system.

The assembly of a JVM out of Jupiter's Source objects is much like the manner in which UNIX command pipelines allow complex commands to be constructed from discrete programs: each program is "instantiated" into a process, and each process is connected to other processes, via pipes, as described by the command syntax; once the process-and-pipe structure has been assembled, data begins to flow through the structure, and the resulting behaviour is determined by the particular choice of programs and their interconnections. Likewise, an incarnation of Jupiter is first constructed by instantiating and assembling Sources. Once the JVM is assembled, the Java program begins to flow through it, like data through the command pipeline. The behaviour of the JVM is determined by the choice of Source objects and their interconnections.

Figure 2 shows part of a typical running incarnation of Jupiter, consisting of interconnected Source objects through which the resources of the executing Java program flow. Also depicted is a typical collection of resource objects. In particular, the `Context` object represents the call stack for an executing Java program. To begin execution, a `Context` is constructed and passed to an `ExecutionEngine`, which sets the rest of the JVM in motion to interpret the program. From the `Context` object, the `MethodBody` object can be reached, which possesses the Java instructions themselves. By interpreting these instructions and manipulating the appropri-

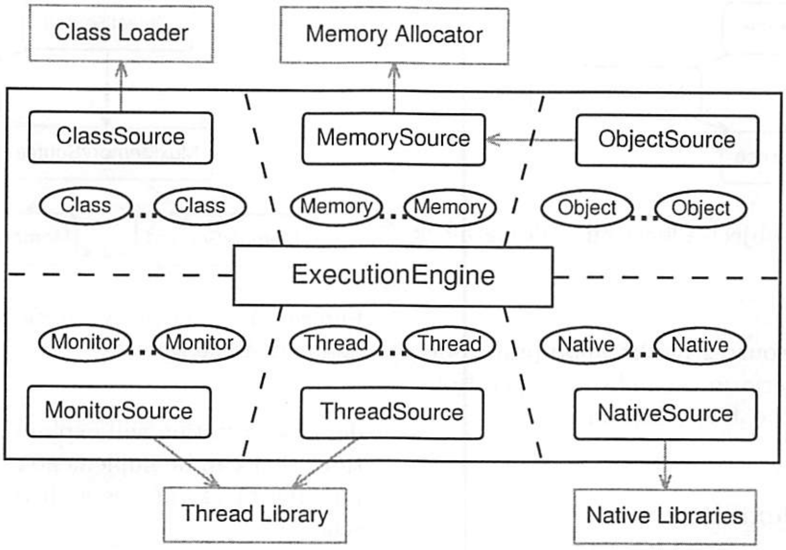


Figure 1: Jupiter's conceptual structure. Resource management responsibility is divided and delegated to a number of Source modules, leaving the `ExecutionEngine` with an "abstract," simplified view of the system's resources.

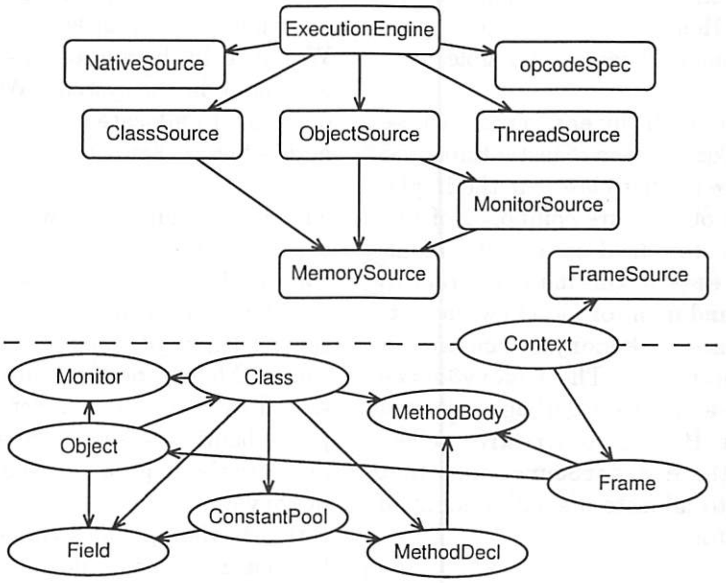


Figure 2: A typical building-block structure. Sources are shown above the dashed line, and resources below. The `Context` object acts as a link between the two when the Java program first begins to execute.

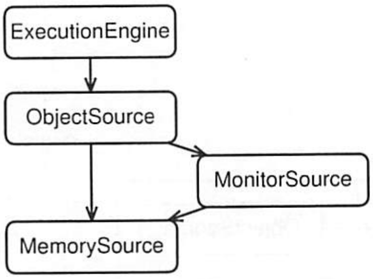


Figure 3: A simple object allocation building-block structure.

ate sources and resources in the appropriate way, Jupiter is able to perform the indicated operations, thereby executing the Java program.

2.1 System Flexibility

In this section, we demonstrate Jupiter's flexibility by examining several configurations of the system's building-block modules. We focus on a recurring example—the object creation subsystem. Through examples, we present several hypothetical ways in which Jupiter could be modified to exploit memory locality on a non-uniform memory access (NUMA) multiprocessor system. In such a system, accessing local memory is less time-consuming than accessing remote memory. Hence, it is desirable to take advantage of local memory whenever possible.

Object creation begins with `ObjectSource`, whose `getObject` method takes a `Class` to instantiate, and returns a new instance of that class. At the implementation level, Java objects are composed of two resources: memory to store field data, and a monitor to synchronize accesses to this data. In order to allocate the memory and monitor for a new `Object`, the `ObjectSource` uses a `MemorySource` and a `MonitorSource`, respectively. The `MemorySource` may be as simple as a call to a garbage collected allocator such as the Boehm conservative collector [12]. Typically, the `MonitorSource` uses that same `MemorySource` to allocate a small amount of memory for the monitor.

The objects employed by such a simple scheme are shown in Figure 3, where arrows indicate the *uses* relation between the modules. The `ExecutionEngine` at the top is responsible for executing the bytecode instructions, and calls upon various facility classes, of which only `ObjectSource` is shown. The remainder of this section will explore the system modifications that can be implemented by reconfiguring the building blocks of this archetypal object allocation scheme.

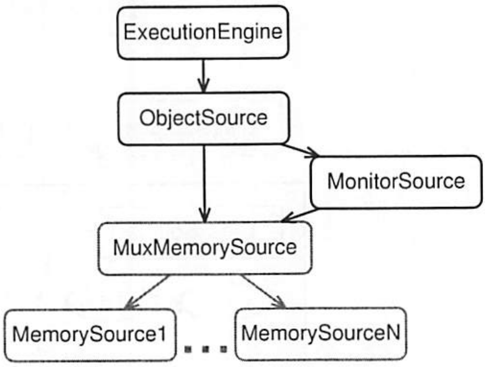


Figure 4: Locality decisions made at the `MemorySource` level.

Suppose the memory allocator on a NUMA system takes a node number as an argument and allocates memory in the physical memory module associated with that node:

```
void *nodeAlloc(int nodeNumber, int size);
```

We can make use of this interface, even though our `getMemory` function of the `MemorySource` facility does not directly utilize a `nodeNumber` argument. We do so by having one `MemorySource` object for each node in the system. We then choose the node on which to allocate an object by calling upon that node's `MemorySource`.

There are a number of ways the `ExecutionEngine` can make use of these multiple `MemorySources`. One way would be to use a "facade" `MuxMemorySource` module that chooses which subordinate node-specific `MemorySource` to use, in effect multiplexing several MemorySources into one interface. This is shown in Figure 4. `MuxMemorySource` uses appropriate heuristics (such as first-hit or round-robin) to delegate the request to the appropriate subordinate `MemorySource`. The advantage of such a configuration is that it hides the locality decisions inside `MuxMemorySource`, allowing the rest of the system to be used without any modification.

A second possibility is to manage locality at the `ObjectSource` level on a per-object basis, as shown in Figure 5. `MuxObjectSource` is similar to `MuxMemorySource`, in that it uses some heuristic to

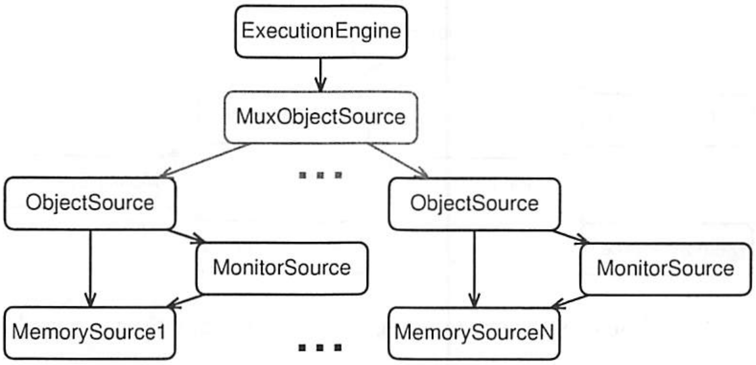


Figure 5: Locality decisions made at the `ObjectSource` level.

determine the node on which to allocate an object. We can use the same node-specific `MemorySource` code as in the previous configuration from Figure 4. We can also use the same `ObjectSource` and `MonitorSource` classes as in the original configuration (Figure 3); we simply use multiple instances of each one. Very little code needs to change in order to implement this configuration.

Yet a third possibility is to allow the `ExecutionEngine` itself to determine the location of the object to be created. Since the `ExecutionEngine` has a great deal of information about the Java program being executed, it is likely to be in a position to make good locality decisions, on a per-thread basis. In this configuration, shown in Figure 6, the `ObjectSource` and `MemorySource` remain the same as in the original configuration. The execution engine chooses where to allocate its objects by calling the appropriate `ObjectSource`. Again, we have not changed `ObjectSource` or `MonitorSource` classes, and the node-specific `MemorySource` class is the same one from the previous configurations.

These examples demonstrate the flexibility of Jupiter's building-block architecture. Each scheme has advantages and disadvantages, and it is not clear which is best. However, the ease with which they can be incorporated allows researchers to implement and compare them with minimal effort.

2.2 Performance Considerations

At first glance, it would appear that our flexible building block structure impairs the performance of the JVM. A researcher who was not concerned with flexibility could simply hard-code the `ObjectSource` to call the `nodeAlloc` function directly. In contrast, our system appears to have two efficiency problems:

- *Call overhead.* Each object allocation request must pass through a number of modules, with each call contributing overhead.
- *Object proliferation.* One node-specific `MemorySource` is required for each node; hence, with hundreds of nodes, hundreds of `MemorySources` will be needed, which would be unnecessary if `ObjectSource` were to call `nodeAlloc` directly.

For a researcher interested in performance, it would be tempting to bypass the module structure entirely, thereby degrading the extensibility of the system.

However, careful exploitation of compiler optimizations allows Jupiter to achieve the performance of the less flexible scheme, without sacrificing flexibility. The reason that this is possible is that each node-specific `MemorySource` is associated with a particular node for the duration of its lifetime, making the node number for each `MemorySource` *immutable.* Immutable data can be freely duplicated without concern for consistency among the multiple copies, since the data never changes. As a result, immutable data that is normally passed by reference can instead be passed by value, with no change to the system's behaviour. This removes the need to dereference pointers, and also eliminates the alias analysis difficulties that make pointer-based code hard to optimize. The system can continue to use the usual abstract, high-level interfaces, and the compiler can produce highly efficient code.

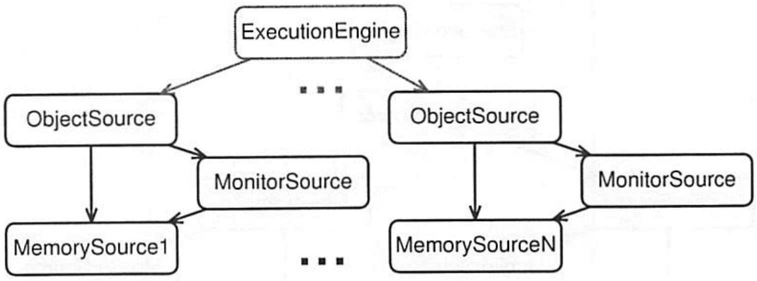


Figure 6: Locality decisions made by the `ExecutionEngine` itself.

To illustrate how this is achieved in our example of object allocation, we begin with the standard Jupiter `MemorySource` interface declarations:

```
typedef struct ms_struct *MemorySource;
void *ms_getMemory(MemorySource this,
                   int size);
```

Because the node number for each `MemorySource` is immutable, it can be passed by value. This can be implemented by replacing the standard declarations with the following:

```
typedef int MemorySource;

static inline void
*ms_getMemory(MemorySource this,
              int size){
  if(this == MS_MUX)
    return nodeAlloc(/* The appropriate
                        node */, size);
  else
    return nodeAlloc(this, size);
}
```

In this version, a `MemorySource` is no longer a pointer to a traditional "heavyweight" object; instead, it is simply an integer representing the node number itself. The `MuxMemorySource` is represented by the special non-existent node number `MS_MUX`. To allocate memory, this code first checks whether the `MuxMemorySource` is being used. If so, it uses the desired locality heuristic to choose a node; otherwise, if a particular node-specific `MemorySource` is used, then memory is allocated from the corresponding node.

With these definitions in place, the existing abstract high-level function calls can be transformed by the compiler into efficient code. Beginning with this:

```
void *ptr = ms_getMemory(
               obs_memorySource(), size);
```

The compiler can perform a succession of function inlining optimizations to produce this:

```
void *ptr = nodeAlloc(/* The appropriate
                         node */, size);
```

Hence, there is no longer any performance penalty for using Jupiter's `MemorySource` interface. This example demonstrates how careful design and implementation allows Jupiter to achieve good performance without any cost to flexibility.

3 System Components and Implementation

In this section, we give a brief tour of the modules which constitute the Jupiter system. The functionality of these modules is exposed through a number of interfaces, known as the *base* interfaces, which are fundamental to Jupiter's design. For each facility provided by Jupiter, we first present its base interfaces, and describe it in terms of the responsibilities it encapsulates. We then describe the current implementation of that facility. It is important to note that these are examples showing the current implementation of the Jupiter facilities. The design of Jupiter allows these implementations to be changed easily.

3.1 Memory Allocation

The `MemorySource` base interface encapsulates the memory allocation facility. It provides just one function, called "getMemory," which takes the size of the memory block required, and returns the resulting block. The current implementation has seven MemorySources, which can be used alone, or in combination, to produce a wide variety of effects:

- MallocMemorySource calls the standard C malloc function to allocate memory. This was useful early in development of the system.
- BoehmMemorySource calls the Boehm conservative garbage collector [12].
- BoehmAtomicMemorySource also calls the Boehm collector, but it marks memory chunks as being pointer-free (that is, *atomic*). This is useful to prevent the garbage collector from unnecessarily scanning for pointers within large arrays of non-pointer data.
- ArenaMemorySource doles out chunks of memory from a given contiguous block, called an *arena.*
- MemoryCounter keeps track of which parts of Jupiter have allocated the most memory. This is useful in debugging to help reduce memory usage.
- Tracer annotates each memory block with information that allows for memory profiling. This is useful to diagnose cases when memory is not being garbage-collected properly, to find out why memory is being retained.
- ErrorMemorySource reports an allocation error when called from specified points within the Jupiter source code. This is useful for testing Jupiter's error handlers, by injecting errors into Jupiter that are otherwise difficult to reproduce.

3.2 Metadata and Method Dispatch

Jupiter creates metadata resource objects to represent the Java program itself. These objects take the form of Classes, Fields, MethodDecls, and MethodBodies. Jupiter accesses classes by name through the ClassSource interface, using a function called getClass. Once a Class has been acquired, its Fields, MethodDecls and MethodBodies can be accessed in order to perform the operations required by the running Java program. A Field encapsulates the data required to locate a field within an object. Typically, it contains nothing more than the field's offset. A MethodDecl encapsulates the data stored in the constant pool to represent METHODREF and INTERFACEMETHODREF entries. A MethodBody encapsulates the data that represents a method implementation; for non-native methods, it holds the bytecode instructions.

Method dispatch is modeled as a mapping from a MethodDecl to a MethodBody. The data structures involved in this mapping are shown in Figure 7. First, the ConstantPool of the target object's Class is consulted to acquire a MethodDecl object. If the MethodDecl has not yet been resolved, the ConstantPool calls upon the Class to locate the MethodDecl, for which it uses a hash table keyed by method name and type signature. As with most JVMs, once the MethodDecl has been acquired, it is cached by the ConstantPool to accelerate subsequent accesses.

Having acquired the MethodDecl, it must now be dispatched to a particular MethodBody. This is done using jump tables indexed by an offset stored in the MethodDecl. Interface methods use a two-level table scheme that allows them to be dispatched in constant time [14]. Once the MethodBody has been acquired, it can be executed, either by interpreting its bytecode, or, for native methods, by invoking the appropriate function through the Java Native Interface (JNI) [11].

3.3 Object Manipulation

Java objects are manipulated through three interfaces: Object, which encapsulates field layout and access; ObjectSource, which encapsulates object allocation and garbage collection; and Field, described above, which encapsulates the data required to locate a field within an object.

These interfaces are implemented as shown in Figure 8, which has similarities to the method dispatch structures shown earlier in Figure 7. The Field abstraction serves an analogous purpose to MethodDecl, representing a field reference in the constant pool. The Field is then passed to the Object, which returns the value of that field.

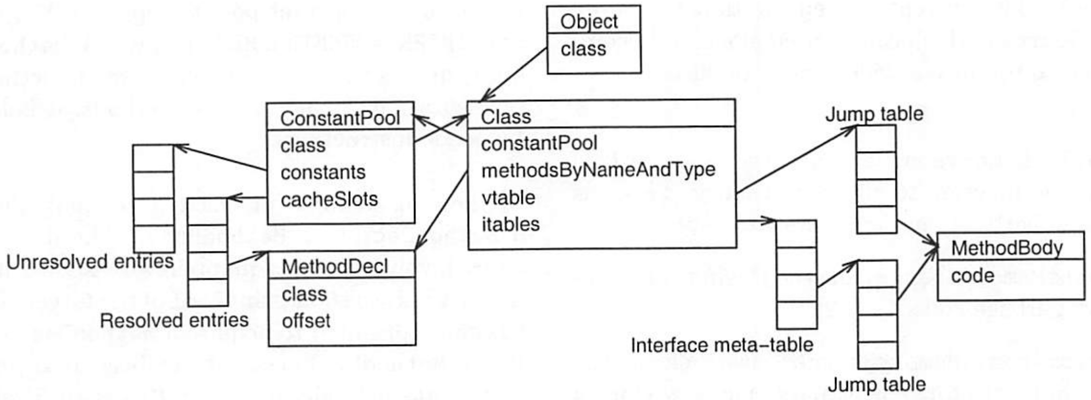


Figure 7: The method lookup objects. Lookup proceeds first as shown on the left to acquire a `MethodDecl`, then on the right to acquire a `MethodBody`.

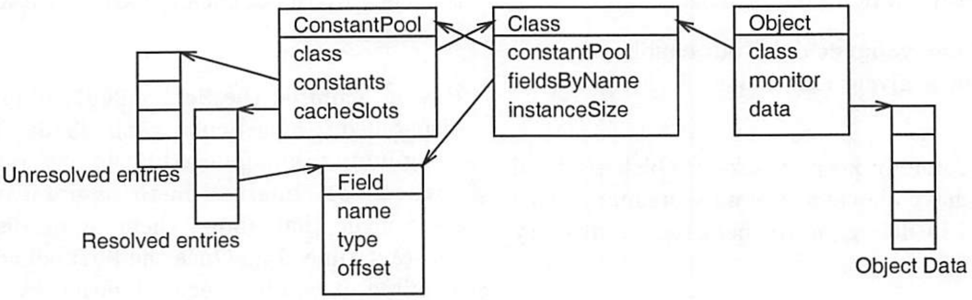


Figure 8: The Jupiter objects responsible for the layout of Java objects.

3.4 Java Call Stack

The call stack of the running Java program is modelled by three interfaces: `Frame`, which encapsulates the data stored in a single stack frame, such as the operand stack and local variables; `FrameSource`, which encapsulates the allocation and layout of `Frames`, controlling such things as the argument-passing mechanism; and `Context`, which encapsulates the storage and management of the call stack as well as the locking logic required by synchronized methods.

Representing the Java context entirely as a data structure allows Java threads to be migrated, simply by executing a given `Context` on a different thread. This stands in contrast to the more straightforward scheme used in Kaffe, which implements method invocation by recursion within the execution engine [9], causing context information to be stored on the native stack, and precluding this kind of migration.

The objects representing the execution stack are shown in Figure 9. The system's view of the stack is provided by the `Context` object on the left, which encapsulates a large array of word-sized *slots* in which the stack contents are stored. Each slot is capable of holding up to 32 bits of data; 64-bit data types require two adjacent slots, as prescribed by the Java specification [15].

Individual stack frames are manipulated through the `Frame` interface, shown as a trapezoid in the figure. A number of design techniques provide the illusion that `Frames` are just like any other objects [16], but in reality, the data for each `Frame` is stored in a contiguous group of slots within the slot array. This allows the frames to be overlapped, making method argument copying unnecessary, while preserving the object-oriented interface. The resulting layout of two adjacent stack frames is shown in the diagram, with overlapping frames labelled twice to indicate their role in each of the frames.

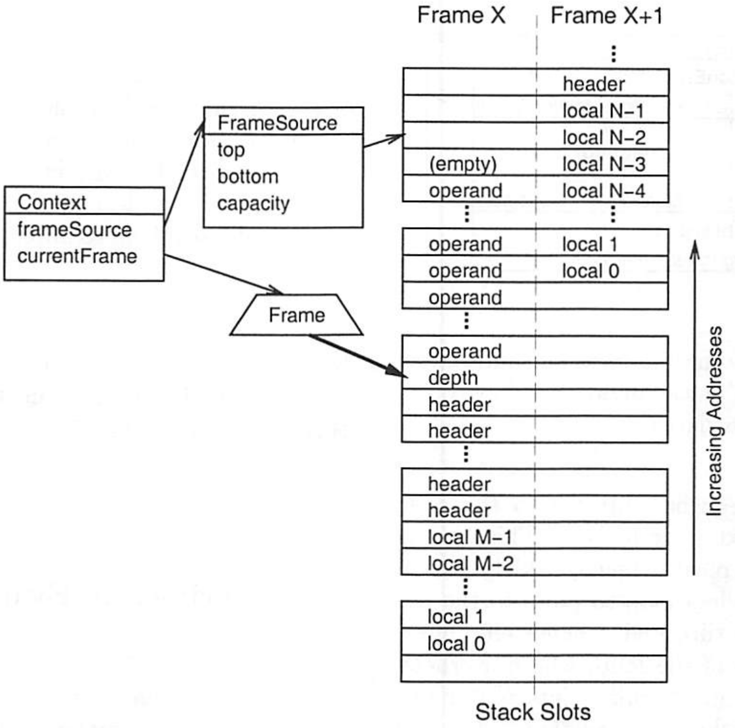


Figure 9: The stack layout. Each stack slot is labelled twice, for its role in the two overlapping frames. The slot marked "(empty)" is the portion of the operand stack space which does not currently contain data.

3.5 Bytecode Interpretation

The ExecutionEngine decodes the bytecode and performs the actions necessary to implement each instruction. Its interface is quite simple, consisting of a single function that takes a Context as an argument, and executes the method whose frame is on top of the Context's call stack.

Since Jupiter's design delegates much of the execution responsibility to other parts of the system, not much remains to be done by the ExecutionEngine itself. The current interpreter implementation divides the functionality into three modules, which are shown along with the ExecutionEngine interface in Figure 10. These modules are each responsible for implementing a portion of the ExecutionEngine functionality:

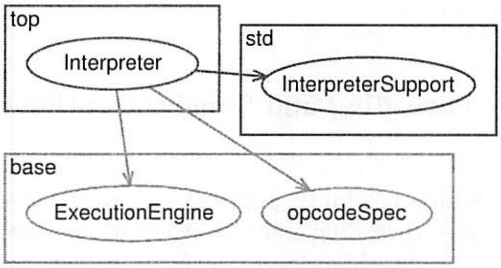


Figure 10: The bytecode execution modules.

- The opcodeSpec module defines each of the Java opcodes in terms of Jupiter's base interfaces. It takes the form of a header file that is included (with #include) into the interpreter module. It is designed to be used by any ExecutionEngine, be it an interpreter or a JIT compiler.
- The InterpreterSupport module provides functionality that is independent of the particular interpreter implementation, such as the stack-unwinding algorithm for exception handling.
- The Interpreter module implements the ExecutionEngine interface, making use of the opcodeSpec and InterpreterSupport modules as necessary.

The current ExecutionEngine implementation is a *threaded interpreter*, meaning that, after execut-

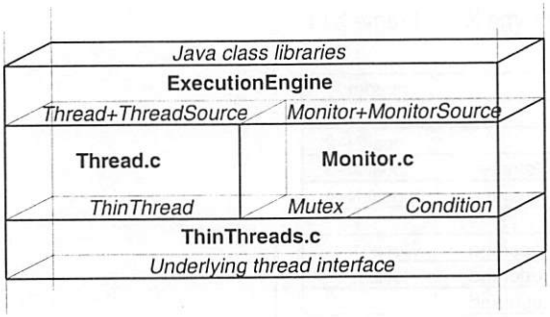


Figure 11: Multithreading modules and interfaces. Modules are shown as blocks divided by horizontal planes representing interfaces.

ing one opcode, it branches directly to the code for executing the next opcode [7]. This stands in contrast to the typical scheme, which uses a `switch` statement inside a loop to jump to the appropriate code. The threaded scheme eliminates the branch to the top of the loop, whose overhead can be substantial in an optimized interpreter like Jupiter's, as will be shown in Section 4.2. The current `ExecutionEngine` also does *bytecode substitution* to improve the performance of `getfield`, `putfield`, `invokevirtual` by dynamically replacing them with faster versions, as will be described in Section 4.2.

3.6 Threading and Synchronization

To maximize flexibility, Jupiter uses two levels of interfaces for threading and synchronization, shown in Figure 11. The high-level interfaces, called `Thread` and `Monitor` (plus the corresponding `Sources`), provide the full Java concurrency semantics. The low-level interfaces, called `ThinThread`, `Mutex`, and `Condition`, provide the minimal semantics required by Java. These low-level interfaces are referred to collectively as the `ThinThreads` interface, which provides a small subset of the POSIX threads semantics [17]. The high- and low-level interfaces are complimentary in several ways:

- `ThinThreads` encapsulates the thread library beneath Jupiter. `Thread` and `Monitor` encapsulate the threading needs of the Java program running on top of Jupiter.
- `ThinThreads` provides the minimal requirements to make implementing Java threads *possible*. `Thread` and `Monitor` provide the maximum support to make implementing Java threads *simple*.
- `ThinThreads` is designed so that the implementation code which connects to the underlying thread library can be trivial. `Thread` and `Monitor` are designed so that the client code which uses them to implement Java threads can be trivial.

Separating the Java concurrency semantics from the semantics of the underlying thread library makes threading and synchronization modules easier to implement and modify.

4 Experimental Evaluation

Jupiter is written in C using an object-oriented style. Although languages such as Java or C++ would provide more support for an object-oriented programming style, and hence more support for our flexible building block architecture, we elected to use C because we did not have confidence in the ability of other languages to deliver good performance. The current implementation comprises approximately 23,000 lines of C code, in about 170 files.

In this section, we quantify the performance Jupiter delivers on standard benchmarks. We also attempt to show the degree of flexibility Jupiter possesses by arguing the ease with which a number of performance optimizations were implemented.

4.1 Overall Performance

To test Jupiter's functionality and performance, we used it to run the single-threaded applications[1] from SPECjvm98 benchmark suite [18]. In this section, we present the execution times consumed by these benchmarks running on Jupiter, and compare them with results from Kaffe 1.0.6, and from the Sun Microsystems JDK v1.2.2-L. We find that Jupiter is faster than Kaffe and slower than JDK.

[1] Although multithreading is already functional in the version of Jupiter we use to report performance, multithreaded performance is currently being optimized. Hence, we elect to report only the performance of single-threaded applications.

	Benchmark	JDK	Jupiter	Kaffe	Jupiter/JDK	Kaffe/Jupiter
1	209_db	178s	282s	836s	1.59:1	2.96:1
2	228_jack	112s	213s	567s	1.91:1	2.66:1
3	201_compress	333s	700s	2314s	2.10:1	3.31:1
4	222_mpegaudio	276s	649s	1561s	2.35:1	2.40:1
5	213_javac	114s	313s	733s	2.74:1	2.35:1
6	202_jess	93s	257s	608s	2.76:1	2.36:1
				Geometric Mean	2.20:1	2.65:1

Table 1: Execution time, in seconds, of the benchmarks using each JVM. The ratios on the right compare the JVMs pairwise, showing the slower JVM's execution time relative to the faster one's.

Table 1 compares the execution times of each benchmark run on the three JVMs. All times were measured on a 533 MHz Pentium III with 512 MB of RAM running Linux, kernel version 2.2.19. Jupiter was compiled with `gcc` version 2.95.2 at optimization level `-O3`, with all source code combined into a single compilation unit to facilitate function inlining [16]. The times were reported by the UNIX "`time`" program, and therefore include all JVM initialization. All benchmarks were run with verification and JIT compilation disabled, since Jupiter does not yet possess either of these features. Averaged across all benchmarks (using the geometric mean), Jupiter was 2.20 times slower than JDK, and 2.65 times faster than Kaffe.

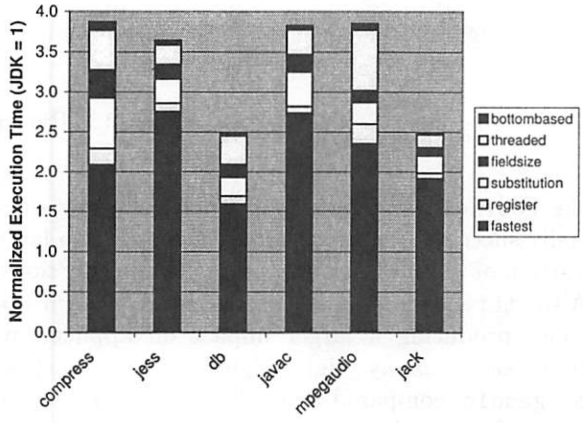


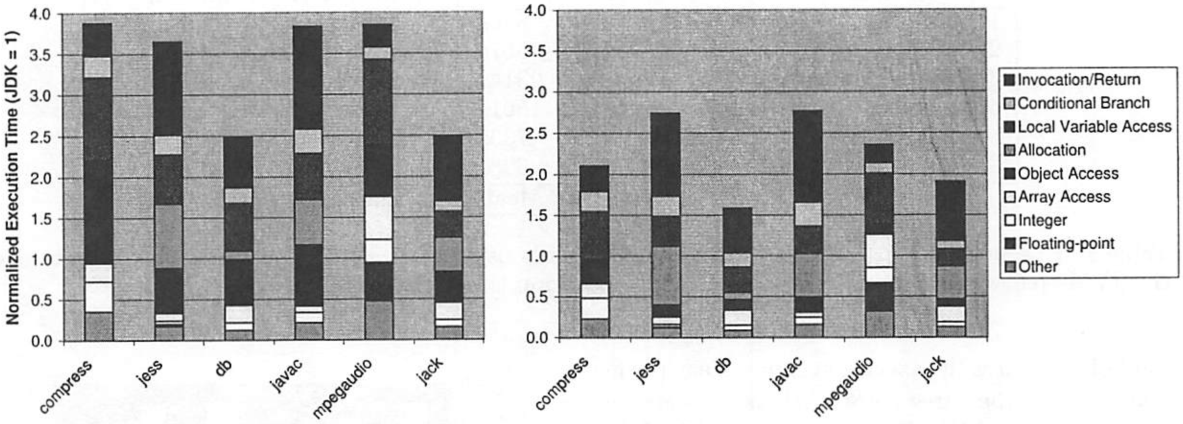
Figure 12: Effect of each optimization on benchmark execution times.

4.2 Performance Optimizations and Analysis

The current level of performance achieved by Jupiter requires that a number of optimizations be implemented. In this section, we briefly describe these optimizations and comment on how the flexible structure of Jupiter facilitated their implementation. The optimizations are:

- *bottombased*: the `Frame` interface was changed to use bottom-based operand stack indexing instead of top-based indexing to speed up stack accesses [16].
- *threaded*: the "loop-and-switch" interpreter was replaced by a threaded version to improve performance, as was described in Section 3.5.
- *fieldsize*: the field's size (either 4 or 8 bytes) was cached inside the `Field` pointer, relieving the interpreter from having to traverse data structures to find this information.
- *bytecode substitution*: the implementations of `getfield`, `putfield` and `invokevirtual` were changed so they replace themselves in the bytecode stream with respective quick versions the first time they execute. These quick versions assume that the field in the opcode has been resolved, and are specialized for the appropriate field size. Furthermore, the opcodes are re-written so that the field offset is stored directly in the bytecode stream, avoiding constant pool access. Subsequent executions of the same bytecode will find the quick instructions, which execute faster.
- *register*: a CPU register was assigned to store the location of the currently executing instruction, eliminating the need to load this value from memory in order to read and execute each instruction.

The impact of these optimizations is shown in Figure 12, which charts the execution-time reduction due to each optimization. The optimizations vary in the degree to which they benefit the performance of each application. For instance, the `fieldsize` and `substitution` optimizations, which targeted



(a) Before optimization

(b) After optimization

Figure 13: Bytecode execution profile.

the `getfield` and `putfield` opcodes, were especially successful on `compress`, which spends a large portion of its time executing these two instructions. Also, `threaded` reduces the overhead of each opcode, producing a larger impact on applications that execute many "lightweight" opcodes, such as `mpegaudio`, compared to applications that execute relatively fewer, but more time-consuming, opcodes, such as `javac` and `jess`.

The above optimizations required minimal effort to implement in Jupiter, attesting to the flexibility of our system. For example, Jupiter's stronger encapsulation confines the modifications required to implement the `fieldsize` optimization to changing a half-dozen lines of code within the `Field` module. In contrast, Kaffe's equivalent of `opcodeSpec` explicitly tests the field type and calls one of a number of "`load_offset`" macros, passing only the field offset as a parameter. To take advantage of cached field size information, all implementations of the `load_offset` macros must be modified to pass the field size in addition to its offset—even those that are not affected by this optimization. Hence, in contrast to Jupiter, the lack of encapsulation within Kaffe causes the scope of this modification to encompass a large number of unrelated modules.

We believe that further improvements to the performance of Jupiter are still possible, which will bring its performance closer to that of the Sun Microsystems JDK. We profiled the amount of time consumed executing each kind of opcode, grouped into the following categories: *Invocation/Return*, *Conditional Branch*, *Local Variable Access Allocation*, *Object Access*, *Array Access*, *Integer*, *Floating-point*, and *Other*. The resulting execution profile is depicted in Figure 13, before and after the optimizations described above. The charts show that our optimization targeted mostly object access overhead, and that more performance-improving opportunities remain. For example, the two slowest benchmarks, `202_jess` and `213_javac`, have similar profiles, with large proportions of invocation/return and allocation opcodes. We will target these opcodes in future work.

Furthermore, Jupiter's coding style relies heavily on function inlining to achieve good performance, so a weakness in the compiler's inlining ability can have a substantial impact. For example, our examination of `gcc`-generated assembly code for `getfield` indicates that the code can be further optimized with nothing more than common subexpression elimination. The exact reason that `gcc` did not successfully perform this optimization is hard to determine. However, it appears that the implementation of `getfield` is too stressful on the inlining facility of `gcc`, hindering its ability to apply the optimization. Applying the optimization manually in the assembly code improves the performance of `getfield` by approximately 15%. The overall performance improvement due to the optimization depends on the application. For example, object access accounts for only 11% of `201_compress`, which would lead to less than 2% overall improvement, and even less for other applications[2]. Similar improvements are possible for other opcodes, and the accumulation of the individual improvements may be substantial.

[2] The small magnitudes of such improvements make them difficult to experimentally measure because they are within measurement error.

5 Related Work

There has been considerable work on improving performance of Java programs in uniprocessor environments [1, 2, 3, 4]. For example, Alpern et al. describe the design and implementation of the Jalapeño virtual machine [1], which incorporates a number of novel optimizations in its JIT compiler. Artigas et al. [3] investigate compiler and run-time support for arrays in Java, and show that improvements can be attained by eliminating runtime checks. Much of this work is orthogonal to ours, in that it improves uniprocessor performance. However, such improvements carry over to multiprocessors, and we expect them to be easily integrated into the Jupiter framework.

There are a number of JVMs and JVM frameworks designed for research into JVM design. They include the Sun Microsystems JDK [8], Kaffe [9], the IBM Jalapeño JVM [1], Joeq [19], OpenJIT [20], and SableVM [7]. However, these frameworks often address flexibility in particular dimensions of JVM design, while in contrast, Jupiter's flexibility is intended to be pervasive and fine-grained, allowing straightforward modification of any aspect of the system. For example, OpenJIT is an object-oriented framework for experimenting with JIT compiler designs and is implemented as a JIT-compiler plugin to Sun's JDK, making it limited to JIT compiler research. Similarly, while Jalapeño (recently released as the Jikes RVM [21]) was designed as "flexible test bed where novel virtual machine ideas can be explored, measured, and evaluated"[1] in an industrial-grade server JVM, much of the work surrounding it explored JIT design. Though its object-oriented design undoubtedly possesses a large degree of inherent flexibility, it is unclear the extent to which the system is flexible in some aspects, such as object layout, stack layout, method dispatch, and so on.

6 Concluding Remarks

In this paper, we presented the design of a modular, flexible framework intended to facilitate research into JVM scalability. We described the building-block architecture employed, as well as the design and implementation of the key modules. Experimentation with our framework demonstrates that Jupiter's flexibility has facilitated a number of modifications, some of which are difficult to accomplish using Kaffe. Measurement of the execution time of the single-threaded SPECjvm98 benchmarks has shown that Jupiter's interpreter is, on average, 2.65 times faster than Kaffe, and 2.20 times slower than Sun's JDK. By providing a flexible JVM framework that delivers good performance, we hope to facilitate our, and others', research into JVM scalability.

Our future work on the Jupiter infrastructure will focus on three main aspects of its implementation. First, we will incorporate a trace-based JIT compiler, called RedSpot, whose implementation is currently underway. Second, we will extend the memory allocation interface in Jupiter to enable the use of a precise garbage collector, and to facilitate the implementation of parallel and concurrent garbage collection. Finally, we plan to modify the manner in which Jupiter stores metadata. At present, Jupiter spreads the responsibility for storing metadata throughout the system, leading to rather heavyweight objects. We believe that by storing the metadata separately, objects will be made lightweight, which would enable further performance optimizations.

References

[1] B. Alpern et al., "The Jalapeño virtual machine," *IBM Systems Journal*, vol. 39, no. 1, pp. 211–238, 2000.

[2] K. Ishizaki et al., "Design, implementation and evaluation of optimizations in a just-in-time compiler," in *Java Grande*, pp. 119–128, 1999.

[3] P. Artigas, M. Gupta, S. Midkiff, and J. Moreira, "High performance numerical computing in Java: Language and compiler issues," in *Proc. of LCPC*, pp. 1–17, 1999.

[4] P. Wu, S. Midkiff, J. Moreira, and M. Gupta, "Efficient support for complex numbers in Java," in *Proc. of Java Grande*, pp. 109–118, 1999.

[5] T. Domani et al., "Implementing an on-the-fly garbage collector for Java," in *Proc. of the Symp. on Memory Management*, pp. 155–165, 2000.

[6] D. Bacon, C. Attanasio, H. Lee, V. Rajan, and S. Smith, "Java without the coffee breaks: A

nonintrusive multiprocessor garbage collector," in *Proc. of PLDI*, pp. 92–103, 2001.

[7] E. Gagnon and L. Hendren, "SableVM: A research framework for the efficient execution of Java bytecode," in *Proc. of USENIX JVM'01*, pp. 27–39, 2001.

[8] Sun Microsystems, http://www.java.sun.com, 2002.

[9] T. Wilkinson, "Kaffe — a virtual machine to run Java code," http://www.kaffe.org, 2002.

[10] "GNU Classpath," http://www.gnu.org/software/classpath/classpath.html, 2002.

[11] "Java Native Interface," http://java.sun.com/j2se/1.3/docs/guide/jni/index.html, 2002.

[12] H. Boehm and M. Weiser, "Garbage collection in an uncooperative environment," *Software Practice and Experience*, vol. 18, no. 9, pp. 807–820, 1988.

[13] O. Krieger and M. Stumm, "HFS: A performance-oriented flexible file system based on building-block compositions," *ACM Trans. on Computer Systems*, vol. 15, no. 3, pp. 286–321, 1997.

[14] F. Siebert and A. Walter, "Deterministic execution of Java's primitive bytecode operations," in *Proc. of USENIX JVM'01*, pp. 141–152, 2001.

[15] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley, 1999.

[16] P. Doyle, "Jupiter: a modular and extensible Java Virtual Machine framework," Master's thesis, University of Toronto, 2002.

[17] IEEE/ANSI, "Posix threads extensions. IEEE/ANSI 1003.1c-1995," 1995.

[18] "SPECjvm98," http://www.specbench.org/osg/jvm98, 2002.

[19] J. Whaley, "joeq virtual machine," http://joeq.sourceforge.net/index.htm, 2002.

[20] F. Maruyama, "OpenJIT 2: The design and implementation of application framework for JIT compilers," in *Work-in-Progress Session of USENIX JVM'01*, 2001.

[21] "The Jikes research virtual machine (RVM)," http://www-124.ibm.com/developerworks/oss/jikesrvm/, 2002.

Trademarks

IBM is a registered trademark of International Business Machines Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Cross-Architectural Performance Portability of a Java Virtual Machine Implementation

Matthias Jacob
Princeton University
mjacob@cs.princeton.edu

Keith Randall
Google, Inc.
keithr@alum.mit.edu

Abstract

This paper describes our experience in porting Compaq's Fast VM from the Alpha processor architecture to the Intel x86 processor architecture. We encountered several opportunities and pitfalls along the way in porting a JVM designed for a RISC architecture to a CISC architecture. Our goal was to preserve most of the FastVM's performance benefits already available on the Alpha platform, and modify or discover new optimizations as they were required for x86. We found that by porting a fast RISC JVM to x86, we could generate a JVM with performance competitive to state-of-the-art production JVM implementations.

1 Introduction

The Alpha processor architecture [23] and the Intel x86 processor architecture [1] have totally different design philosophies. Alpha, which is a RISC architecture [18], provides a minimal, simple instruction set which can be efficiently decoded. Intel x86 is a CISC architecture which is designed to run more complex operations within a single instruction, and thus includes more different instructions and formats. While porting Compaq's Fast VM [6] from Alpha to x86, we encountered several opportunities and pitfalls because of this change in architectural philosophy.

Fortunately, many parts of the JVM required little or no modification when switching from one architecture to another. These parts include the class loader, bytecode verifier, and most of the garbage collector. Other parts of the JVM were ported by others - we took advantage of Sun's port of the Java libraries to x86/Linux so we did not have to repeat that work. Instead, we concentrated on the major changes required in the just-in-time (JIT) compiler and closely related modules, like the stack unwinding mechanism. Crucial to a successful (i.e., fast) port of the JIT was maintaining the quality of generated code that was the result of many optimizations performed in the RISC JIT. We found that some optimizations were straightforward to port, other optimizations required major rework, and still others were simply unworkable in a CISC architecture. Finally, we also found that some additional optimizations not required at all by a RISC machine were of critical importance to fast CISC code.

The different design philosophies of the Alpha and x86 architectures impose different design constraints on a Java virtual machine:

- *Reduced number of registers:* The Alpha architecture has 31 registers, compared to the x86 architecture which has only 8. This differential makes it crucially important to do register allocation well on the x86.
- *Instructions contain multiple operations:* On a RISC architecture instructions either load values from memory, store values into memory, or execute arithmetic operations. In contrast, CISC architectures support complex instructions that integrate these different RISC functions into a single instruction. Selecting the optimal instruction for a certain task, therefore, becomes more difficult on the x86.
- *Different addressing modes:* Because x86 instructions decompose into multiple operations, similar instructions are built from slightly different primitive operations. For example, an addition can add a value in memory and a value in a register, or add two registers.
- *Non-orthogonality of instruction set:* Not all registers can be used with every instruction,

so CISC architectures impose additional constraints on how data is allocated to registers.

- *Source registers get overwritten:* Within an arithmetic instruction, a source register is often overwritten on a CISC architecture to store the result. If the old value of the source register is needed, an additional copy step before such an instruction is required.

In addition to these five general aspects, the x86 architecture has the following design differences with Alpha:

- *32-bit architecture:* Porting the JVM from a 64-bit architecture to a 32-bit architecture introduces several complications. Since the Java VM supports 64-bit integer operations, a 32-bit implementation must emulate these operations using multiple instructions. Furthermore, the 32-bit architecture limits the maximum feasible heap size to 4GB.

- *Limited set of registers per instruction:* RISC instructions support access to either all integer or floating-point registers, depending on the instruction. On the x86 architecture, certain instructions require their arguments to be in certain registers. For example the shift operations require the shift amount to be given in register %cl, whereas in the Alpha architecture, the shift amount can be in any register. These restrictions impose additional complexity on register allocation [9].

- *Floating-point stack versus floating-point registers:* In the x86 architecture, all floating-point operations are executed on a floating-point stack instead of floating-point registers. Operationally, an arithmetic operation on two floating-point values pops the first two elements on the floating-point stack, executes the operation, and pushes the result on the floating-point stack. The resulting stack has one less element than the original stack. The register allocator must take these movements into consideration.

- *Floating-point precision toggle:* On the Alpha architecture, the precision of the floating-point operation is always encoded in the instruction itself, whereas it needs to be explicitly set by an additional instruction in the x86 architecture before the instruction operates on two registers on the floating-point stack.

The following two sections describe various optimizations we implemented in the x86 JVM. Section 2 describes modifications we made to existing optimization algorithms to port them from Alpha to x86. Section 3 describes new optimizations implemented specifically for the x86.

2 Modified Optimizations

The Fast VM for Alpha [6] is a fast, full-featured virtual machine for Alpha and Tru64 Unix. For the purposes of this paper, we will concentrate on the JIT inside this JVM, as the majority of modifications required to port to x86/Linux occur inside the JIT. In this section, we describe three JIT optimizations that are crucial for efficient performance on x86 and which, for various reasons, required some redesign for the x86 port.

We describe the optimizations here – we postpone our experimental analysis of these optimizations to the performance section (Section 4).

2.1 Register Allocation

A crucial difference between the x86 and Alpha architectures is the number of directly-addressable machine registers. In particular, the x86 architecture has only 8 integer registers (none of which are completely general-purpose), compared to 31 on the Alpha. This differential makes it crucially important to do register allocation well on the x86. Many algorithms for register allocation have been developed for use in standard compilers [11, 10, 12, 14, 21]. Although these allocators generate excellent allocations, they are typically too slow for use in a just-in-time setting. Much recent work has focused on faster just-in-time allocators [19]. The Alpha register allocator, which we adapted for use on the x86, does a simple global[1] allocation based on access frequency, and then uses a greedy allocator within each basic block.

The data structure used for the allocator in the FastVM is called an *lmap* ("location map") (see Figure 1). In the global allocation phase, each Java *entity* (Java stack location S_i or local variable L_i) is assigned a *home* machine location (H), either a register or stack slot. The register allocator dedicates every register either as a home location (H) for a particular Java entity, or a temporary location (T)

[1] *global* in this context means across a single procedure.

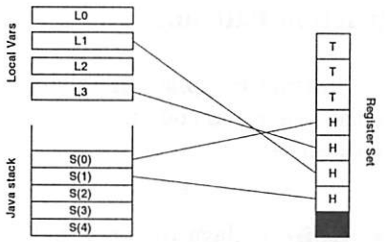


Figure 1: LMAP register allocation.

that can be used for storing intermediate results or call arguments. The allocation of home locations is a simple priority-based allocator, where the priority of a Java entity is just its estimated access frequency. Because interference analysis is expensive, the allocator assumes that every Java entity interferes with every other Java entity. The lack of interference analysis is not too detrimental as Java entities, particularly stack slots, encompass many live ranges. At each basic-block boundary, all live entities are forced to their home locations.

Within a basic block, the allocator uses a greedy allocation algorithm. The lmap maintains a mapping from Java entities to the locations where their values are currently stored to facilitate allocation. Methods of lmap are used to move arguments into registers, allocate additional temporary registers, and record the result registers. The lmap maintains separate allocation information for integer and floating-point registers.

Since the number of registers on the x86 is relatively small, it is important to use those registers during code generation effectively. We implemented a number of optimizations of the RISC register allocation algorithm to improve register utilization. First, the RISC allocator divides the available register set into two categories - home locations and temporary locations. For the x86 allocator, such a static partitioning was too restrictive, so we allowed the allocator to use home locations as temporaries when the home location was dead. This allowed us to increase the percentage of registers that are allowed to be home locations without overly constricting the temporary register set. Second, the x86 instruction set is not orthogonal - not all instructions can use all registers, so additional code was added to the allocator to allow allocation from a specific subset of possible registers. Finally, many instructions in the x86 architecture can accept arguments in memory locations instead of registers, so the allocator was also modified to understand that allocation of particular arguments to registers is optional, so that under high register-pressure conditions the memory form of the operation is used.

Wasteful static register allocations had to be eliminated as well. Because the Alpha architecture has lots of registers, the Alpha JVM assigned a few registers exclusively to particular tasks. One register was dedicated to holding a pointer to thread-local state, one register was reserved for interface method invocation, and one register was reserved as a scratch location for the code generator. Dedicating 3 registers to specific uses is tenable when 31 registers are available, but when only 8 registers are available, being carefree with your registers is not advisable.

Of particular importance to the performance of the RISC JVM is fast access to thread-local storage. The object allocator and mutex mechanism make heavy use of thread-local variables, so access to them must be fast. The RISC JVM assigned a register to point to the thread's own variables, so access to those variables required just a load or store with this register as a base. The CISC JVM cannot afford to spare a general-purpose register for this purpose. However, the x86 has other registers, one of which we "stole" to provide fast thread-local access without requiring any general-purpose registers. We store our thread-local pointer in a segment register on the x86 to avoid using a general-purpose register (fortunately, the %fs segment register is unused by current Linux x86 compilers and libraries).

For the other two purposes that the RISC JVM reserves a register, we instead reserve a thread-local variable. Although typically slower than a register, the thread-local variables have the same semantics as a register and thus can be used in their stead. We also take advantage of the fact that most instructions that use these reserved registers in the RISC JVM have analogues in the x86 world that accept our thread-local memory locations (constant offsets from %fs) instead of registers. The CISC nature of the x86 is thus a two-edged sword - fewer registers require us to be crafty about our register use, but at the same time the x86 provides us with more instructions and addressing modes to be crafty with.

The combination of these register allocation improvements increased the JVM's spec rating by 68%. For details on our experiments, see Section 4.

2.2 Instruction Selection

Instructions on a RISC architecture are structured in a systematic way. They can be categorized into ALU operations, memory operations, and control operations. ALU operations always work on registers, and every register from the register set can be addressed by every ALU operation. Memory operations move values between the register set and memory. This structure makes instruction selection relatively easy, and the Java data types map 1:1 to the instruction architecture.

The optimal selection of instructions is more complex on x86 than it is on Alpha for the following reasons:

- *Different addressing modes:* Because a lot of operations exist in different addressing modes, unlike a RISC processor, the correct kind of instruction needs to be chosen in order to avoid any additional instructions for moving values. For example if the values for an addition operation are in a memory and in a register the instruction selection algorithm always picks an add instruction that adds a memory and a register location.

- *Limited set of registers per instruction:* When picking the next instruction, the code generator always checks in which registers the current values are in and chooses the instruction appropriately. If the current register allocation doesn't fit to the instruction at all, values need to be moved. This scheme could be improved with more global analysis, but at the expense of a larger compile-time cost.

- *Efficient 64-bit operations:* The Java bytecode contains 64-bit integer and floating-point operations that the x86 platform needs to support. For each of these bytecode operations the number of temporary registers and the amount of memory accesses need to be minimized. For example, the following code is one possible implementation of the `ladd` (64-bit integer addition) bytecode instruction.

```
mov     0x0(%esp,1),%eax
add     0x8(%esp,1),%eax
mov     0x4(%esp,1),%ecx
adc     0x10(%esp,1),%ecx
```

2.3 Instruction Patching

Because just-in-time compilation takes place in a single pass, the generated code needs to be fixed up in certain situations:

- *class initializers:* Java requires that a class be initialized before any of its methods or fields are accessed. For any reference in the code we are compiling that refers to an uninitialized class, the JIT must insert instructions before the reference to ensure that the class is initialized. After the class is initialized, these added instructions are superfluous and can be patched to NOPs for improved performance.

- *fix up branches:* When generating code, the target address of forward branches is not yet known, so these branches must be fixed up when the destination address is known.

- *copying registers:* In some situations, register copy operations can be avoided by renaming registers backward in the already-generated code.

- *inlining:* Small methods can be directly inlined at the call site.

Since the instruction length is fixed on RISC architectures, it is relatively easy to implement instruction patching efficiently and safely.

In certain cases (see Section 2) instructions need to be patched at runtime. Patching in a JVM can be a delicate operation, because often the code that is being patched is being executed concurrently by another thread or processor. Thus, all intermediate states of the patching operation need to be seen by all processors as valid and correct code.

On Alpha, patching is straightforward since every instruction is exactly 4 bytes long. Each instruction can be atomically replaced with a single write (and the JVM is designed to only require single-instruction patching). However, on the x86 architecture, instruction lengths may be different, which complicates the patching process. We need to make sure that we can atomically patch any instruction of reasonable length with another, possibly different length, instruction sequence.

To make patching possible, at code generation time we ensure that any patchable instruction contains enough bytes for any instruction which we might want to patch over it, and that the patch location is suitably aligned so that an atomic operation can be used to perform the switch. On Pentium-class processors, this means that the patch location must not straddle a cache-line (32-byte) boundary. Finally, at patch time we equalize the lengths of the sequences by padding the inserted sequence with nop's, and use an atomic compare-and-exchange operation to ensure the patch is performed exactly once.

2.3.1 *Inlining*

Small compiled methods that fit into a call site can easily be inlined using the code patching mechanism. The JVM inlines a method by patching the method body into a call site and pads the remaining space with nops. Interestingly, inlining in this manner allows us to inline a method *after* all call sites to it have been compiled. This kind of inlining typically happens because inlining at the time we compile the caller is not always possible, usually because of class initialization constraints. More sophisticated techniques for inlining can be found in [13].

2.3.2 *Retargeting*

The Alpha JIT uses *retargeting* to avoid register-to-register copies. When a Java entity needs to be moved from one register to another, the Alpha JIT scans backward in the code to find the instruction that generated the value (if it exists), and rewrites it to use the new destination.

Although retargeting is very effective on the Alpha, it is difficult to implement on the x86 for three reasons. First, it is difficult in general to walk backward in code when instructions are not fixed size. Second, many x86 instructions generate their results in particular registers and thus are not retargetable. Finally, for many x86 instructions the output register is also an input register, so rewriting the output register alone is not possible. For these reasons, we added instead a forward-looking heuristic into our register allocator. The allocator computes the preferred register into which each Java entity should be placed, based on the requirements of its nearest future use. The allocator then uses these suggested registers, if possible, to satisfy allocation requests. By preferentially using the suggested registers, the allocator often ensures that the entity is in the correct location when the use of the entity is later encountered.

register	use	type
eax	1st int arg, first int return	scratch
ecx	method ptr	scratch
edx	2nd int arg, second int return	scratch
ebx		preserved
esi		preserved
edi		preserved
ebp		preserved
esp	stack pointer	preserved

Figure 2: Optimized Calling Convention

3 Optimizations for x86

Simply updating the JVM optimizations to take into account the different properties of x86 was not enough, however. The x86 platform has additional peculiarities that required the implementing of some additional optimizations. We describe these optimizations here.

3.1 Calling Convention

There are four problems with the x86 calling convention that make it difficult to port the RISC JVM to x86. First, the x86 argument passing and (some) return value passing are done on the stack instead of in registers. Second, the x86 dedicates two registers to stack management, a frame pointer and a stack pointer, when it is possible to use only a single register. Third, we need to be able to unwind the stack to implement the Java exception model, and changes to the x86 calling convention were required to simplify and speed up this unwinding. In addition, Java requires precise detection of stack overflow, which is difficult in the standard calling convention because almost any instruction can cause a stack overflow. Finally, the x86 calling convention enforces only 4-byte alignment of stack frames, which can be a performance problem because 8-byte stack operations might be unaligned.

In order to solve all of these problems, we developed a new calling convention as shown in Figure 2. This register assignment gives us 3 scratch (caller-save) registers and 4 preserved (callee-save) registers, plus a stack pointer.

We modified the calling convention to use a fixed stack pointer over the life of a method, as opposed to the standard x86 convention which encourages the use of push and pop instructions which modify the stack pointer. Local stack variables can be accessed at constant offsets from the stack pointer. The optimized stack scheme of our implementation is shown in Figure 3. The prolog/epilog and a sample callsite of the optimized calling convention can be found in Figures 4 and 5 respectively.

By allocating a callee-saved register slot at the bottom of the stack frame, the prolog of a method can immediately check whether a stack overflow has occurred by storing a callee-saved register (or any value, if there aren't any registers that need to be saved) to the bottom of the stack frame. Thus, the only instructions that can cause a stack overflow are the first store in the method prolog, and call instructions (which push their return address). At both of these locations, stack overflow exceptions are simple to deal with.

We also took the opportunity while changing the calling convention to align stack frames to 8-byte boundaries for faster stack operations on the `double` type.

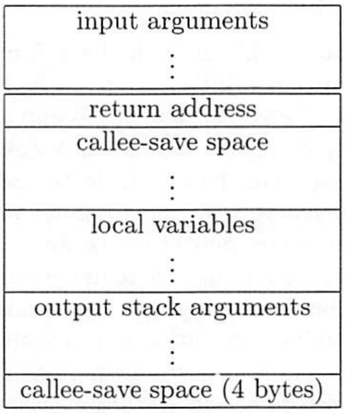


Figure 3: Optimized stack frame layout

```
subl $24, %esp
movl %ebx, (%esp)  # save %ebx,
                     stack check
...body of method...
movl (%esp), %ebx  # restore %ebx
addl $24, %esp
ret
```

Figure 4: Method prolog/epilog of the optimized calling convention.

```
movl $1, %eax     % 1st arg
movl $2, %edx     % 2nd arg
movl $3, 4(%esp)  % 3rd arg
call method
                  % return value
                    in %eax
```

Figure 5: Example call site in the optimized calling convention.

3.2 Floating-Point Modes

Switching the floating-point precision mode on x86 inserts an additional instruction into the code and also causes stalls in the processor pipeline. For Java methods using 32-bit and 64-bit floating-point operations, switching precision is inevitable. However, our experiments have shown that while running the benchmarks, only one method in the Java class libraries and one method in the benchmark programs (mpegaudio) use two precision modes within a single method. Even in these cases the less frequent precision is used at most two times. To take advantage of this fact, we choose a default precision mode for each method and set the precision mode to the default at the beginning of each method that uses floating-point operations. Thus, default precision operations require no additional instructions, while the precision mode needs to be set and reset around the occasional non-default precision operation. Our calling convention considers the precision mode to be a preserved value, so it must be reset at the end of the method.

Because non-default precision operations are rare, this strategy significantly decreases the number of switches required and thus increases performance. Precisely analyzing the control-flow and optimizing the number of switches in a given method would help, but may be too expensive for just-in-time compilation.

4 Performance

In this section we show performance results for the FastVM implementation on x86 and compare it to other state-of-the-art JVM implementations on x86. The second part presents improvements of the different single optimization methods which we have explained in the previous sections.

To compare the different JVM implementations on x86 we ran the SpecJVM98 [4] benchmark suite (large size) on a Compaq Deskpro machine (Pentium III 866MHz) with 256MB main memory running Linux 2.4.3. The heap size of the JVM is 128MB since we want to avoid as many side effects of garbage collection activity as possible during the measurements. Setting the heap size to 128MB eliminates most garbage collections.

For performance measurements we set up a Java wrapper program that invokes every single benchmark three times. At the end we compare the geometric mean as well as the best and the worst of all three benchmark times.

With a just-in-time compiler the first run usually takes the most time and any succeeding run is shorter since later runs invoke methods that have already been compiled to native machine code. We found that after three runs most JVM implementations do not improve much further by using already precompiled code. We also chose to restart the JVM after a single benchmark, so that benchmarks cannot take advantage of a method that has been compiled by a benchmark invoked earlier.

The FastVM uses a simple heuristic approach to decide whether a method needs to be compiled or not. During execution the FastVM counts how often a method gets invoked, and if the count exceeds an upper limit the methods gets compiled and optimized. In contrast, in a feedback-based compiler, optimization takes place only in the parts of the code that are frequently executed. To find out which parts the JVM needs to optimize, the compiler gets program profiling information as feedback from the runtime system. The advantage of feedback-based systems is that frequently executed parts of the program get well optimized. On the other hand, profiling and optimizing code imposes an additional runtime penalty and often code can be efficiently optimized using lightweight optimization methods as described in this paper.

4.1 Performance comparison of different JVM implementations

In this performance evaluation we compare the FastVM to Sun's JRE 1.3.1 (HotSpot client and server) and IBM JRE 1.3.0. These two JVM implementations are the leaders in the SpecJVM benchmark results and therefore a reasonable performance indicator.

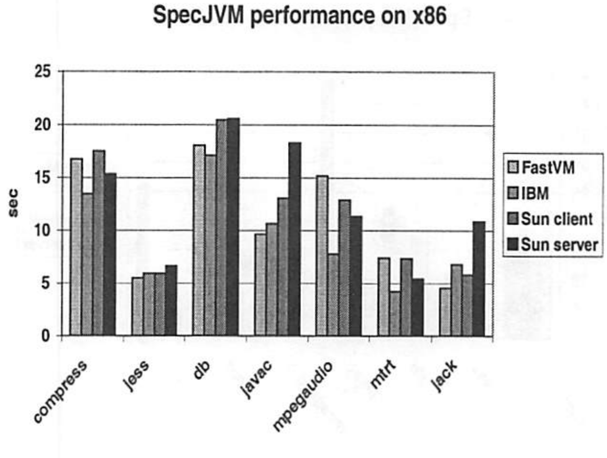


Figure 6: Average runtimes of SpecJVM on x86.

Figure 6 shows the average runtime of the four JVMs over three runs. The FastVM is the fastest JVM on jess, javac, and jack, and comes last in mpegaudio and mtrt. Mpegaudio and mtrt mainly use floating point operations, and so far we have not spent much effort in optimizing floating point operations apart from switching precision modes as explained in section 3.2. The FastVM is faster by 5-30% on some benchmarks and can be slower as much as 95% in mpegaudio.

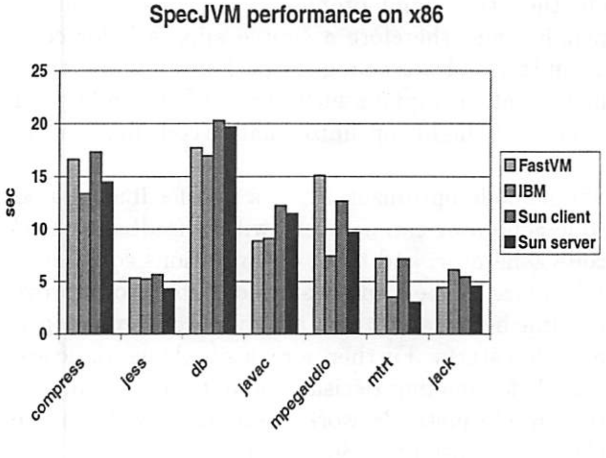


Figure 7: Best-case runtimes of SpecJVM on x86.

Figure 7 shows the best-case scenario. Here the feedback-based JVM code generators have an advantage since they can spend more time on optimizing code. However, the FastVM is still among the fastest JVMs for jess, jack, and javac. The lead of the FastVM is only around 2-3%.

In the worst-case scenario (Figure 8) feedback-based compilers, especially the Sun Hotspot server, lag behind since they spend a lot of time during the

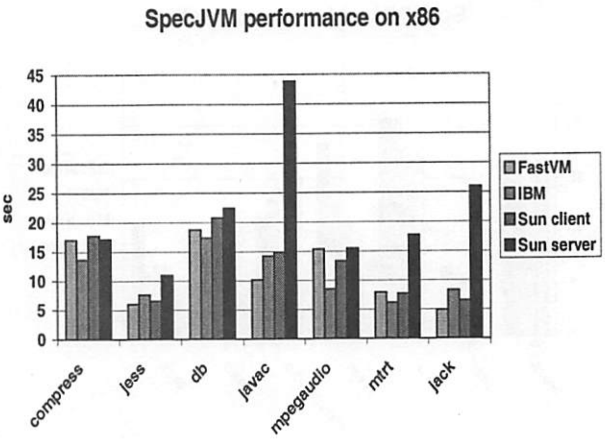


Figure 8: Worst-case runtimes of SpecJVM on x86.

first run in order to optimize native code. Here the FastVM has a lead in jess, javac, and jack. In javac the FastVM can be up to 39% faster than other implementations.

It is difficult to compare feedback-based code generators and non-feedback-based code generators. If a Java method that is difficult to optimize gets called frequently it is better to use a feedback-based compiler that applies heavy optimization techniques. On the other hand profiling code adds a runtime penalty, and therefore a simple approach for code optimization leads to a good performance. Furthermore, static compilers such as Swift [20] can be used to generate highly optimized native code in advance.

Simple code optimizations of a non-feedback-based code generator can compete with a feedback-based code generator, and these optimizations could even take place in the feedback-based system to improve runtime before any strong and time-consuming optimization starts. Furthermore, a simple heuristic approach for making decisions about whether or not to compile methods works reasonably well on the client applications of SpecJVM.

4.2 Optimizations for the FastVM on x86

This section investigates the effects of the different code optimizations we implemented for the FastVM on x86. These optimizations are register allocation, method inlining, and floating-point precision mode optimizations as we described in section 3. All three optimization techniques impose a negligible amount of additional compilation time to the VM.

4.2.1 Register Allocation

In this section we test the performance of the optimized register allocation scheme, and furthermore, how the benchmarks behave when we reduce the number of available registers.

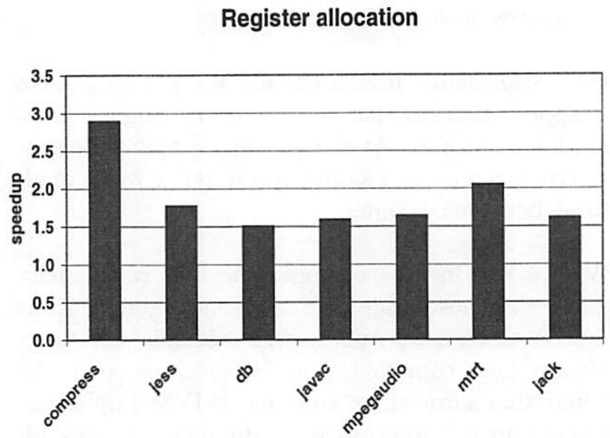


Figure 9: Benchmark speedup when enabling register allocation. The performance number represents the ratio between the measured runtime with the non-optimized compiler and the optimized compiler running register allocation only.

Figure 9 shows the overall performance improvement when we use register allocation instead of copying values from and to memory at each operation. The maximum speedup factor we achieve by enabling register allocation is about 3 for compress, but generally every benchmark profits from passing the arguments in registers.

The reason that compress gets more speedup than the other benchmarks is that it frequently runs sequential operations within one method that can be optimized well.

This effect becomes clearer in Figure 10 where we decrease the number of registers to a minimum of three and measure the performance. The y-axis indicates the speedup to the same code optimization that uses only three registers. In benchmarks that profit a lot from register allocation like compress we get a gradual speedup from three to seven available registers. In other benchmarks performance may even slow down a bit. The reason is that by using our simple local register allocation scheme we do not always pick the theoretically optimal allocation, and therefore performance may slow down by a nuance if the register allocation that uses only one register less was closer to the optimal solution. Benchmarks

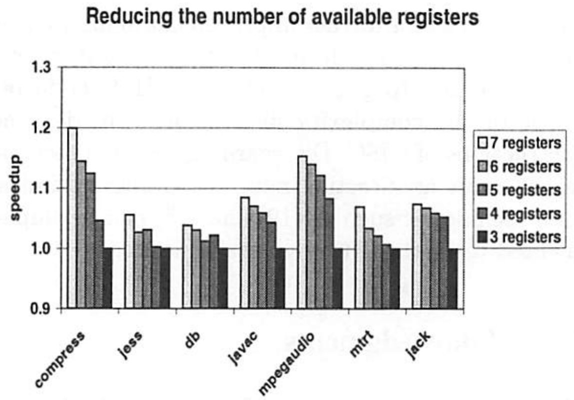


Figure 10: Reducing the number of available registers in the register allocator. The performance number represents the ratio between the measured runtime with four, five, six, and seven registers and the measured runtime with three registers.

that show only little improvement when we increase the number of available registers mostly profit from passing method arguments in registers.

4.2.2 *Method inlining*

This section shows the impact of inlining small methods. Before register allocation takes place we inline short methods before register allocation takes place. Figure 11 shows that inlining of short methods generally has only a marginal influence on performance for most benchmarks. Javac is the only exceptions with a speedup of 1.15.

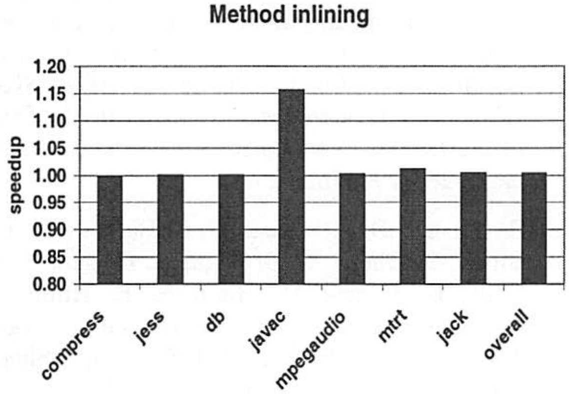


Figure 11: Benchmark speedup for inlining of short methods. The performance number represents the ratio between the measured runtime with method inlining enabled and the measured runtime with method inlining disabled.

4.2.3 *Floating-Point precision toggle*

When we set the floating-point precision at every single floating-point instruction explicitly, a significant number of memory operations gets added to the instruction stream that stalls the processor pipeline and thus degrades performance. By using a simple heuristic for determining whether a method uses single or double precision predominantly (as described in section 3), we are able to increase performance by a factor of up to 1.8 as shown in Figure 12.

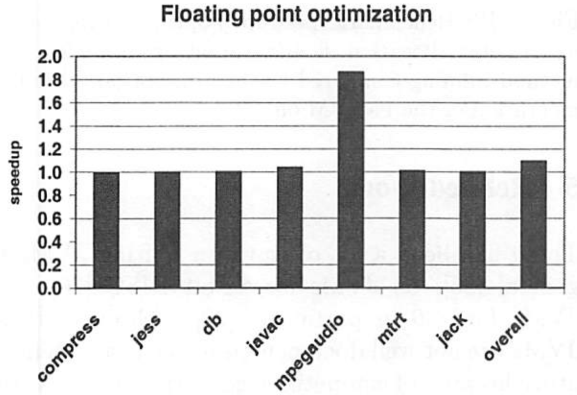


Figure 12: Benchmark performance when enabling optimization for toggling floating-point precision. The performance number represents the ratio between the measured runtime with floating-point mode optimization enabled and the measured runtime with floating-point mode optimization disabled.

In SpecJVM, only mpegaudio and mtrt use floating point operations frequently. However, in mpegaudio the compiler generates more register to register floating point operations that require the precision mode to be switched.

4.2.4 *Overall performance improvement*

Figure 13 shows the complete picture of the optimized code generator versus a non-optimized naive code generator.

Benchmarks profiting most from the optimizations are compress and mpegaudio. Each of them improves by a different optimization technique, which demonstrates the necessity of multiple optimization passes. The optimization techniques are simple and inexpensive and can be widely applied.

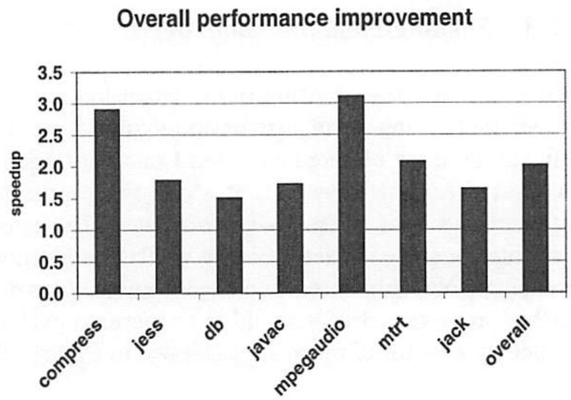


Figure 13: Benchmark speedup of optimized version using register allocation, floating-point optimization, and method inlining compared to the non-optimized implementation of the FastVM on x86.

5 Related Work

There has been a lot of work on writing JVMs in general [25], on JVMs for 64-bit RISC [16], and JVMs for x86 in particular [22]. Many of these JVMs are not well documented in the research literature because of competitive concerns. Two notable exceptions are IBM's Jalapeño JVM[8] and Intel's Open Research Platform(ORP)[7]. IBM's JVM is similar to ours in that it started out life as a RISC JVM on PowerPC, and has recently been ported to x86 [24]. Intel's JVM is of course native to the x86, but may experience similar porting pains when moving the IA64[2]. [17] describes the architecture of the HotSpot Virtual Machine. [6] describes the FastVM for Alpha that we ported to x86. In addition to just-in-time compilers several static compilers exist that generate native code from Java source code and use standard compiler backends for optimization [3, 5, 20] Unlike much of the published literature on x86 JVMs, our work focuses on the gritty, low-level design considerations required to make a JVM use a CISC instruction set effectively.

6 Conclusion

We have shown that it is possible to port a JVM implementation from a 64-bit RISC architecture to a 32-bit CISC architecture spending minimal effort and without losing much performance. The achieved performance is competitive with state-of-the-art Java Just-in-time compilers. Nevertheless, there were some pitfalls to get around, including floating-point precision mode, register allocation, and calling convention. On the other hand we had opportunities for further improvement in instruction selection. After all, from our experience it is generally simpler to generate efficient RISC code because of the complexity in addressing modes and instructions of CISC. Disregarding the architectural issues, more architecture-neutral compiler optimization techniques such as [19] and [15] can be implemented to further improve performance.

7 Acknowledgments

We thank the people at SRC, especially Rustan Leino and Mike Burrows, for help with implementation and insightful discussions on Java bytecode and architecture issues.

References

[1] *IA-32 Intel Architecture Software Developer's Manual.*

[2] *Intel Itanium^{TM}Architecture Software Developer's Manual.*

[3] NaturalBridge, BulletTrain. `http://www.naturalbridge.com.`

[4] SpecJVM98. `http://www.spec.org/osg/jvm98.`

[5] Toba. `http://www.cs.arizona.edu/sumatra/toba.`

[6] The Compaq Fast Virtual Machine, June 1999. `http://www.compaq.com/java/FastVM.html.`

[7] A.-R. Adl-Tabatabai, M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth. Fast, effective code generation in a just-in-time java compiler. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI'98)*, pages 280–290, June 1998.

[8] B. Alpern, D. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM System Journal*, 39(1), Feb. 2000.

[9] A. Appel and L. George. Optimal spilling for CISC machines with few registers. *ACM SIGPLAN Notices*, 36(5):243–253, May 2001.

[10] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):428–455, May 1994.

[11] G. J. Chaitin. Register allocation & spilling via graph coloring. In *SIGPLAN '82 Symposium on Compiler Construction*, pages 98–105, 1982.

[12] F. C. Chow and J. L. Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, Oct. 1990.

[13] D. Detlefs and O. Agesen. Inlining of virtual methods. In R. Guerraoui, editor, *Proceedings ECOOP'99*, volume 1628 of *LNCS*, pages 258–278, Lisbon, Portugal, June 1999. Springer-Verlag.

[14] L. George and A. W. Appel. Iterated register coalescing. *ACM transacations on programming languages and systems.*, 18(3):300–324, May 1996. also in POPL'96.

[15] M. Kawahito, H. Komatsu, and T. Nakatani. Effective null pointer check elimination utilizing hardware trap. *ACM SIGPLAN Notices*, 35(11):139–149, Nov. 2000.

[16] A. Krall and R. Grafl. CACAO – A 64 bit JavaVM just-in-time compiler. *Concurrency: Practice and Experience*, 9(11):1017–1030, 1997.

[17] M. Paleczney, C. Vick, and C. Click. The Java HotSpot™Server Compiler. In *Proceedings of First Usenix Java™ Virtual Machine Research and Technology Symposium*, Apr. 2001.

[18] D. A. Patterson. Reduced instruction set computers. *Communications of the ACM*, 28(1):8–21, Jan. 1985.

[19] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, Sept. 1999.

[20] D. J. Scales, K. H. Randall, and S. G. J. Dean. The Swift Java Compiler: Design and Implementation. Technical Report 2, Compaq Western Research Laboratory, April 2000.

[21] R. Sethi and J. D. Ullman. The generation of optimal code for arithmetic expressions. *Journal of the ACM*, 17(4):715–728, 1970.

[22] K. Shudo. shuJIT.
`http://www.shudo.net/jit`.

[23] R. L. Sites and R. L. Witek. *Alpha AXP architecture reference manual.* Digital Press, 12 Crosby Drive, Bedford, MA 01730, USA, second edition, 1995.

[24] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

[25] B.-S. Yang, S.-M. Moon, S. Park, J. Lee, S. Lee, J. Park, Y. C. Chung, S. Kim, K. Ebcioğlu, and E. Altman. LaTTe: A Java VM just-in-time compiler with fast and efficient register allocation. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT '99)*, pages 128–138, Newport Beach, California, Oct. 12–16, 1999. IEEE Computer Society Press.

An Empirical Study of Method Inlining for a Java Just-In-Time Compiler

Toshio Suganuma   Toshiaki Yasue   Toshio Nakatani

IBM Tokyo Research Laboratory
1623-14 Shimoturuma, Yamato-shi, 242-8502 Japan
{suganuma, yasue, nakatani}@jp.ibm.com

ABSTRACT

Method inlining is one of the optimizations that have a significant impact on both system performance and total compilation overhead in a dynamic compilation system. This paper describes an empirical study of online-profile-directed method inlining for obtaining both performance benefits and compilation time reductions in our dynamic compilation system. We rely solely on the profile information in deciding which methods should be inlined along which call paths, without employing the existing static heuristics, except that methods with extremely small bodies are always inlined. The call site distribution and invocation frequency are collected using dynamically generated instrumentation code, and are given to the recompilation controller for analyzing and organizing the inlining requests. The experimental results show that the strategy of relying on the profile information in method inlining decision is quite effective for reducing compilation overhead and/or for improving the performance. It can also provide flexibility for inlining decisions based on the type of profile, such as spiky or flat profiles. The results show that the strategy can be the basis of automatic optimization selection for a future efficient dynamic compilation framework.

1. INTRODUCTION

Dynamic compilation systems need to reconcile the conflicting requirements between fast compilation speed and fast execution performance. We would like the system to generate highly efficient code for good performance, but the system needs to be lightweight enough to avoid any startup delays or intermittent execution pauses that may occur due to the runtime overhead of the dynamic compilation.

The high performance implementation of Java Virtual Machines (JVM) is moving toward exploitation of adaptive compilation optimizations on the basis of online runtime profile information [1][9][20][23]. Typically these systems have multiple execution modes. They begin the program execution using the interpreter or baseline compiler as the first execution mode. When the program's frequently executed methods or critical hot spots are detected, they use the optimizing compiler for those selected methods to run in the next execution mode, obtaining better performance. Some systems employ several different optimization levels to select from based on the invocation frequency or relative importance of the target methods.

In these systems, the set of optimizations provided for each optimization level is predetermined. Basically those optimizations considered to be lightweight, such as those with linear order to the size of the target code, are applied in the earlier optimization levels, and those with higher costs in compilation time or greater code size expansion are delayed to the later optimization levels. However, the classification of optimizations into several different levels has been either intuitive work or just based on the measurements of compilation cost and performance benefit on a typical execution scenario analyzed offline [2]. Consequently, the optimizations applied are not necessarily effective for the actual target methods compiled with the given optimization level. That is, some optimizations may not be able to contribute to any useful transformation for performance improvement, but can result in a waste of compilation resources.

The problem here is that we equally apply the same set of optimizations for those methods selected to compile at that optimization level, regardless of the type and characteristics of each method. Ideally it is desirable to dynamically assemble a set of suitable optimizations depending on the characteristics of the target methods so that we can apply only those optimizations known to be effective for the method. It would be better for the total cost and benefit management, if we could not only selectively apply optimizations on performance critical methods, but also selectively assemble the optimizations that are effective for those methods.

Method inlining, a well-known and very important technique in optimizing compilers, expands the target procedure body at the method invocation call sites, and it defines the scope of the compilation boundary. Toward the big picture of the efficient compilation framework, method inlining can naturally be the first target for a dynamic compilation strategy by exploiting the profile information collected at runtime, since it is one of the op-

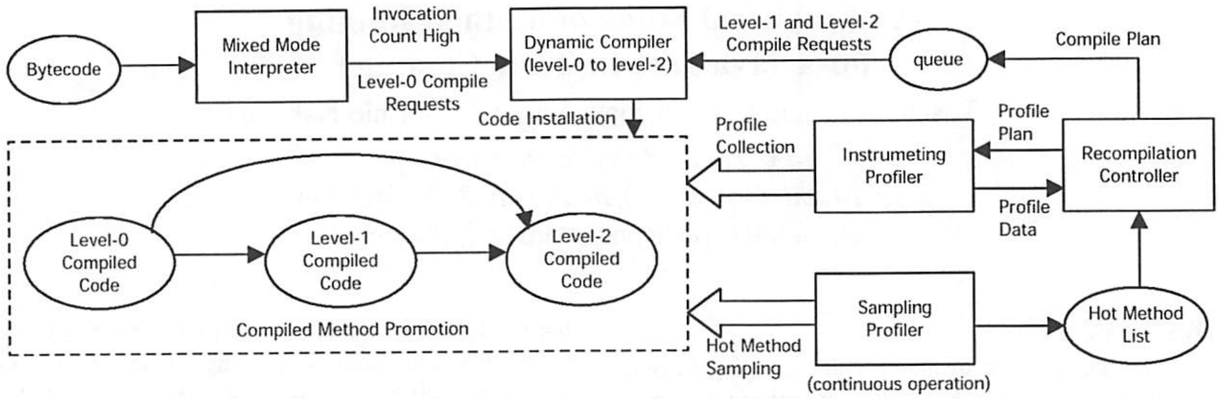


Figure 1. System architecture of our dynamic compilation system.

timizations that have a significant impact on both performance and compilation overhead.

In this paper, we describe an empirical study of an online-profile-directed method inlining strategy in our dynamic compilation system. We rely solely on the profile information for deciding which methods should be inlined along which call paths. Though many static heuristics exist, the only one we use is to always inline very small methods. The ultimate goal of our research is to explore the opportunities for dynamic automatic optimization selections, and the first step is to evaluate the potential benefits and limitations of this approach for method inlining.

The following are the contributions of this paper:

- **Online collection of call site information:** We present a mechanism for dynamic instrumentation, which collects the call sites distribution and invocation frequency for hot methods in order to help make decisions on method inlining during recompilation. This is quite different from the existing approach of constructing a dynamic call graph through sampling during the entire period of program execution.
- **Detailed evaluation of profile-directed method inlining policies:** We present detailed evaluation results of the benefits and limitations regarding both performance and compilation overhead for several inlining policies: two variations of profile-directed method inlining, existing static inlining heuristics, and a hybrid of both. We also evaluated inlining based on offline-collected profile information to know the upper bound of the potential of the profile-directed method inlining.

The rest of this paper is organized as follows. The next section is an overview of our dynamic compilation system, and describes the existing inlining policy and potential impacts on both performance and compilation time. Section 3 describes our design and implementation of online-profile-directed method inlining using the instrumentation-based information about call site. Section 4 presents the experimental results on both performance and compilation overhead by comparing several inlining policies. Section 5 discusses the results and possible future research towards automatic optimization selection. Section 6 summarizes the related work, and finally Section 7 presents our conclusions.

2. BACKGROUND

The overall system architecture of our dynamic optimization framework is described in detail in [23]. In this section, we briefly describe the system characteristics and features that are closely related to this study, and then we describe the existing static-heuristics-based inlining policy and its impact on both compilation time and performance.

2.1 System Overview

Figure 1 depicts the overall architecture of our system. This is a multi-level compilation system, with a mixed mode interpreter (MMI) and three compilation levels (level-0 to level-2). Initially all methods are interpreted by the MMI. A counter for accumulating both method invocation frequencies and loop iterations is provided for each method and initialized with a threshold value. The counter is decremented whenever the method is invoked or loops within the method are iterated. When the counter reaches zero, the method is considered as frequently invoked or computation intensive, and the first compilation is triggered.

The dynamic compiler has a variety of optimization capabilities. The level-0 compilation employs only a very limited set of optimizations for minimizing the compilation overhead. For example, it considers method inlining only for extremely small target methods as described in Section 2.2. It disables most of the dataflow optimizations except for very basic copy and constant propagation. The level-1 compilation enhances level-0 by employing additional optimizations, including more aggressive full-fledged method inlining, a wider set of dataflow optimizations, and an additional pass for code generation. The level-2 compilation is augmented with all of the remaining optimizations available in our system, such as escape analysis and stack object allocation, code scheduling, and DAG-based optimizations[1].

The level-0 compilation is invoked from the MMI and is executed as an application thread, while level-1 and level-2 compilations are performed by a separate compilation thread in the background. The upgrade recompilation from level-0 compiled code to higher-level optimized code is triggered on the basis of the hotness level of the compiled method as detected by a timer-based sampling profiler [24]. Depending on the relative hotness level, the method can be promoted from level-0 compiled code to either level-1 or directly to level-2 optimized code. This decision is made based on different thresholds on the hotness level for each level-1 and level-2 method promotion.

The sampling profiler periodically monitors the program counters of application threads, and keeps track of methods in threads that are using the most CPU time by incrementing a *hotness count* associated with each method. The profiler keeps the current hot methods in a linked list, sorted by the hotness count, and then groups them together and gives to the recompilation controller at every fixed interval for upgrade recompilation. The sampling profiler operates continuously during the entire period of program execution to adapt effectively to the changes of the program's behavior.

There is another profiler, an instrumenting profiler, used when detailed information needs to be collected from the target method. The instrumenting profiler, when invoked, dynamically generates code for collecting specified data from the target method, and installs it into the compiled code by rewriting the entry instruction of the target. After collecting a predetermined amount of data, the generated instrumentation code automatically uninstalls itself from the target code in order to minimize the performance impact. Thus the compiler can take advantage of the online profile information dynamically collected at runtime for the level-1 and level-2 compilations. This instrumentation mechanism is used for collecting call site distributions and execution frequencies to guide method inlining in this study.

This dynamic instrumentation mechanism allows the intensive monitoring of the runtime application behavior for a certain interval of the program's execution. By combining this technique with the sampling profiler, we can benefit from two nice properties of online profiling: low overhead and reliable profiling results. It is low overhead because we can limit the targets of the instrumentation to only those hot methods we are interested in for recompilation. It is reliable because we can obtain representative profiles for real applications by delaying the instrumentation installation until the program hot regions are discovered.

2.2 Static Heuristics Based Inlining

Our dynamic compiler is able to inline any methods regardless of the context of the call sites and the types of caller or callee methods. For example, there is no restriction in inlining synchronized methods or methods with try-catch blocks, nor against inlining methods into call sites within a synchronized method or a synchronized block. Thus the inlining decision can be made purely from the cost-benefit estimate for the method being compiled.

There is a class of methods in Java that simply result in a single instruction of code when compiled, such as the one just returning an object field value. Other methods may turn out to be very small number of instructions for their procedure body after compilation. We call these *tiny methods*. Our implementation identifies them based on the estimated compiled code size[2].

***Tiny method*:** This is a method whose estimated compiled code is equal or less than the corresponding call site code sequence (argument setting, volatile registers saving, and the call itself). That is, the entire body of the method is expected to fit into the space required for the method invocation.

Since invocation and frame allocation costs outweigh the execution costs of the method bodies for these methods, and since inlining them is considered completely beneficial without causing any harmful effects in either compilation time or code size expansion, they are always inlined at all compilation levels. Otherwise we employ static heuristics to perform method inlining in an aggressive way while keeping the code expansion within a reasonable size.

The inliner first builds a possibly large tree of inlined scopes with allowable sizes and depths using optimistic assumptions, and then looks at the total cost by checking each individual decision to come up with a pruned final tree. When looking at the total cost, the inliner manages two separate budgets proportional to the original size of the method: one for tiny methods (and profile-directed methods, if any), and the other for any type of method. It tries to greedily incorporate as many methods as possible in the given tree using the static heuristics until the predetermined budget is used up. Currently the static heuristics consist of the following rules.

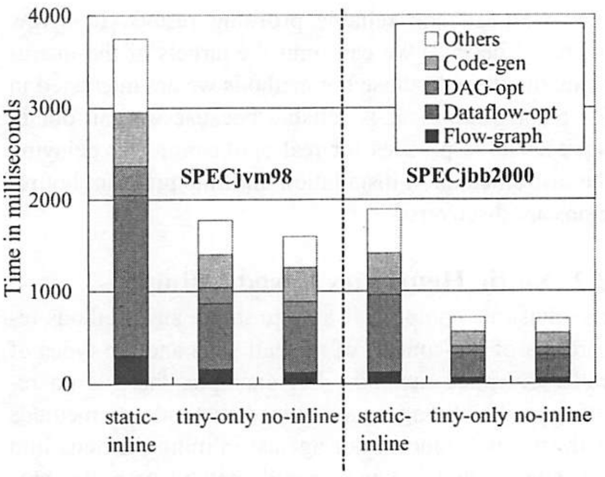


Figure 2. Compile time breakdown for SPECjvm98 and SPECjbb2000 benchmarks with three different inlining policies: static-inline, tiny-only, and no-inline.

- If the total number of local variables and stack variables for the method being compiled (both caller and callees) exceeds a threshold, reject the method for inlining.
- If the total estimated size of the compiled code for the methods being compiled (both caller and callees) exceeds a threshold, reject the method for inlining.
- If the estimated size of the compiled code for the target method being inlined (callee only) exceeds a threshold, reject the method for inlining. This is to prevent wasting the total inlining budget due to a single excessively large method.
- If the call site is within a basic block which has not yet been executed at the time of the compilation, then it is considered a cold block of the method and the inlining is not performed. On the other hand, if the call site is within a loop, it is considered a hot block, and the inlining is tried for deeper nest of call chains than for outside a loop.

The devirtualization of dynamically dispatched call sites is done at all compilation levels based on the class hierarchy analysis (CHA) and type flow analysis, and it produces either guarded code (via class tests or method tests) or unguarded code (via code patching on invalidation) [15]. Preexistence analysis is also performed to safely remove guard code and back-up paths without requiring on-stack replacement [11]. The resulting devirtualized call sites then may or may not be inlined according to the static rules described above.

Only tiny methods are inlined in level-0 compilation. In level-1 and level-2 compilation, the broader range of methods within the conditions of the above static heuristics are considered for inlining. There is no difference between level-1 and level-2 compilation in terms of the scope and aggressiveness of inlining.

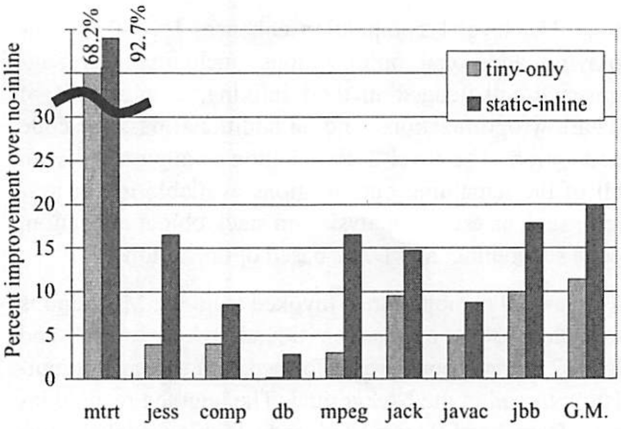


Figure 3. Performance difference for SPECjvm98 and SPECjbb2000 benchmarks with static-inline and tiny-only inlining policies over the no-inline policy.

As a result of method inlining, a data structure called an inlined method frame (IMF) is produced for each call site to provide information about the inlining context for the runtime system, which needs to know the exact call stack and call context prior to inlining. For example, the security manager requires the correct depth of the current call-chain on the stack, or a runtime exception is raised. The exception handler also requires the inlining context of call sites in order to keep track of the handlers for catching exceptions from the current context.

Figure 2 shows the compilation overhead for three different policies of method inlining when running two industry standard benchmarks. *Static-inline* indicates using the existing static heuristics for inlining in level-1 and level-2 compilation, *tiny-only* means inlining only tiny methods in level-1 and level-2 as well as level-0 compilation, and *no-inline* performs no method inlining at any level of compilation regardless of the size of the target methods. Note that devirtualization is still enabled for all three cases. The data was collected based on the system configuration described in Section 2.1 and using the methodology described in Section 4.1.

Each bar gives a breakdown of where in the optimization phase time is spent for compilation. *Flow-graph* indicates the time for basic block generation, inlining analysis to determine the scope of the compilation boundary, and flow graph construction for the expanded inlined code. *Dataflow-opt* is the time for a variety of dataflow-based optimizations, such as constant and copy propagation, redundant null-pointer and array-bound check elimination. *DAG-opt* is for DAG-based loop optimization and pre-pass code scheduling. *Code-gen* includes the time for register assignment, code generation, and code scheduling. *Others* denotes memory management cost and any other code transformations, including live analysis and peephole optimizations, each costing less than 5% of the total compilation time. Figure 3 shows the performance differences between the same set of three inlining policies.

As we can see in these figures, method inlining has significant impact on both compilation time and performance. With the static-inline policy, the time spent for inlining analysis and flow graph construction itself is not a big part of the total overhead, but optimizations in later phases, dataflow optimizations in particular, cause a major increase of the compilation time due to the largely expanded code that has to be traversed. The code size expansion and the work memory usage also have similar increases with this inlining policy, although the data is not shown here due to space limitations.

The tiny-only inlining policy, on the other hand, has very little impact on compilation time, as expected, while it produces significant performance improvements over the no-inline case. For some benchmarks, it provides a majority of the performance gain that can be obtained using the static-inline policy. On average, it contributes a little over half of the performance improvement compared to the static-inline policy. From these figures, our current policy of always inlining tiny methods is clearly justified.

3. PROFILE-DIRECTED INLINING

Method inlining can significantly impact both costs and benefits of compilation as shown in the previous section. We would like to apply this expensive but effective optimization selectively, rather than statically, based on dynamic program execution behavior, only for program points that are expected to provide the performance benefits. We basically remove any existing static heuristics for method inlining as described above, except for always inlining tiny and small sized methods, and totally depend on the online profile data. Section 3.1 describes how the profile data can be collected for extracting potential candidates for method inlining, and Section 3.2 provides the procedure for identifying exact call paths to make inlining requests before recompilation. Section 3.3 describes the problem of handling small methods, and Section 3.4 discusses how to make inlining decisions more appropriate for flat profile applications.

3.1 Collection of Call Site Information

As described in the previous section, we exploit the dynamic instrumentation mechanism for collecting information on the call site distribution and execution frequency at runtime. The instrumentation is basically applied for all hot methods that are detected by the sampling profiler and which are candidates for promoting to the next level of optimization. Since the instrumentation code in this case is to find the most beneficial candidates among the call sites where the current target method can be inlined, we need to ensure whether the current target is appropriate for inlining, or otherwise performing instrumentation is just overhead. Thus those methods that are apparently not inlinable into other methods (due to excessively large size, for example) are excluded as targets for instrumentation. Methods already compiled with the highest level of optimization are also excluded, since they are methods already considered for inlining but not inlined in the previous round of compilation.

As shown in Figure 4, the code generated by the instrumenting profiler is dynamically installed in the target code by replacing the entry code (after copying it into the instrumentation code region) with an unconditional jump instruction to direct control to the instrumentation code. At its first entry, the instrumentation code stores the current time stamp. It then examines the return address stored on the top of the current stack, and records it in a form of table with values and their corresponding frequencies. Because of the overhead, only the single-level history in the call stack is recorded. The maximum number of entries in the table is fixed. When the predetermined number of samples has been collected, the code again checks the current time and records how long it took to collect those samples by subtracting the start time. The code then uninstalls itself from the target code by restoring the original code at the entry point[3].

Thus two kinds of information can be collected for each method during an interval in the program's execution: the distribution of callers and the frequency of invocations. The time recorded for sample collection indicates call frequency for the method for that fixed number of instrumentation samples. Since the hotness-count provided by the sampling profiler can largely depend on the size of the target method (it is easier for larger methods to get more sampling ticks), small but very frequently called methods may not receive the attention they deserve. In that sense, the fine-grained frequency information can be useful in identifying hot call edges critical for inlining, and in compensating for the rather coarse-grained hotness count of the sampling profiler.

The instrumentation is successively applied to a group of recompilation target methods provided by the sampling profiler at every fixed sampling interval. After installation, the recompilation controller periodically monitors the completion of profile collections on all of these methods, and if completed, it proceeds to the inlin-

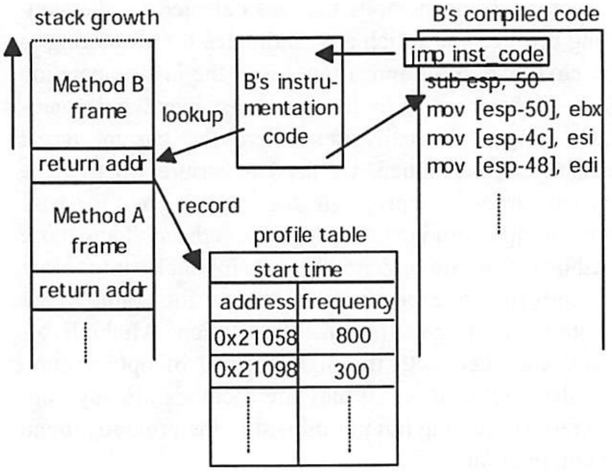


Figure 4. Dynamic instrumentation for collecting information on call site distribution and execution frequency (after code patching).

ing analysis step described in the next section. This is implemented by setting a maximum wait count for the recompilation controller to give up the thread execution. If necessary, it stops profiling for methods that are invoked very infrequently and which will never reach the predetermined number of samples by directly manipulating the current profile count for those methods.

3.2 Profile-Based Inlining Decision

The recompilation candidates grouped together and given to the recompilation controller are then examined to identify the hot call paths appropriate for inlining among them. Upon completion of collecting the call site information, the decision on requesting method inlining proceeds with the following steps.

1. **Examination of the call site distribution profile:** For each of the instrumented methods, the profiling result is examined in the first step. If there are just one or a few dominant call sites found in the profile information and the execution frequency is not too low, these are considered important candidates for inlining and thus a connection is created between the caller and the callee for each hot call edge. The edge is associated with a distribution ratio based on the count given to the corresponding call site address in the profile[4]. At the end of this step, a partial call graph is constructed, where nodes indicate recompilation candidate methods, each of which has a hotness count, and edges show dominant call connections from caller to callee, each assigned a distribution ratio (see Figure 5).

2. **Identifying exact call paths appropriate for inlining:** In the next step, we identify the exact call paths for inlining requests by traversing the connection between multiple methods. We begin with a method having no outgoing edge, and traverse the connections up to the top of the call chain. In the traversal of the connections, the following two operations need to be done:

   Call site address conversion: Since the raw data of the profile information indicates a set of compiled code addresses where the invocation was made to the current target method, it does not specify the exact call sites in the original form of the method bodies. The compiled code address is therefore converted to the offset of the corresponding bytecode instructions using the table generated at compile time.

   Hotness count adjustment: This is to correct the relative hotness level by assuming that inlining for the connected methods is performed. A portion of the hotness count of the callee (including values from its children) is distributed to each of its callers according to the ratio for each incoming edge. When dividing hotness counts from children into their multiple callers, the distribution ratio cannot be known precisely, because the profile information is not available for multiple-level histories of the call paths. Therefore, we use a *constant ratios assumption* [21] to approximate the actual ratio. That is, we assume the ratio is the same as the one recorded in the profile.

3. **Request for method inlining:** Finally, inlining requests are made for the hot call paths identified above. These requests are stored in a persistent database so that inlining decisions can be preserved for future recompilation to incorporate the previous inlining requests. This is important, as described in both the Self-93 [14] and Jalapeño [1] systems, to

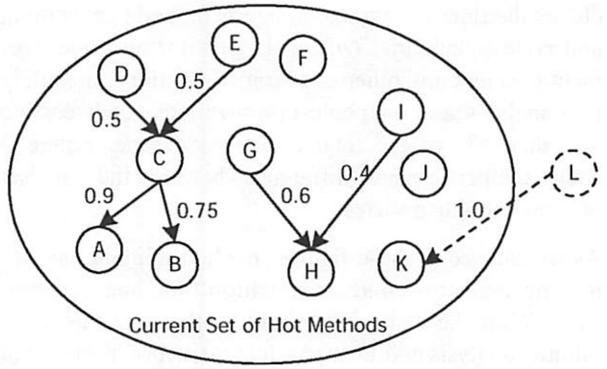


Figure 5. A simple graph is constructed for the profile-based inlining decisions. Nodes indicate methods to be recompiled, and edges show the dominant call edges from caller to callee. These dominant call edges are likely to be included within the hot call paths that should be considered for inlining requests.

avoid a performance perturbation resulting from oscillating between two compiled versions of a method, each with a different set of inlining decisions.

4. **Recompilation request:** After all the relationships between the hot methods in the given group are examined and appropriate inlining is requested, the list of hot methods is again traversed to issue the final recompilation requests. If a method in the list is requested to be inlined into another method, and there is no other call site for this method found in the profile information, then the method is removed from consideration for recompilation. Likewise, those methods whose hotness count is below a threshold value as a result of the hotness count adjustment are discarded. A method already compiled with the highest optimization level will be recompiled again only when a new inlining request has been made and its estimated impact is above a threshold value.

Since we consider the inlining possibilities among hot methods appearing in the group of hot methods, and since this group comes from sampling during an interval in the program execution, the resulting inlining requests are expected to contribute for a performance boost. In Step 2, however, there may be a call path where a caller is not included in the current group of hot methods, and it is possible to request inlining for such a case as well, as shown by the dashed line circle in Figure 5. This can be considered to be more aggressive inlining, and is evaluated in Section 4 as a variation of our profile-directed inlining. We do not trigger the recompilation for the caller in this case, since the method is not known to be worth upgrading to higher optimization at this time. We leave such a method for the future until it is be promoted by the sampling mechanism, at which time the inlining requests previously made for this method will all be incorporated in the recompilation.

3.3 Small Methods

Inlining with the above mechanism is only possible when the target method is identified as hot for dynamic instrumentation. Although tiny methods are always inlined, there are still some chances that other methods with very small bodies will be missed for selection as recompilation candidates due to the sampling nature of the profiler, and in that case it is unable to consider the inlining possibilities for those methods. To remedy this problem, small methods are also inlined, in addition to tiny methods, in level-1 and level-2 compilation under profile-directed inlining.

It is difficult to define small methods. In contrast with tiny methods, inlining them may cause some additional overhead for both compilation time and code size expansion. The timer interval of the sampling profile affects which small methods may be overlooked as inlining candidates. Our goal here is to spot as many of these overlooked candidates as possible, but also to minimize additional compilation overhead that can result from statically inlining these methods. In our current implementation, small methods are defined such that the estimated compiled code is less than the total invocation overhead occurred in both caller and callee (the method prologue and epilogue, in addition to the call site code sequence itself).

3.4 Flat Profile Applications

Benchmarks and applications can be roughly categorized into two different sets, each having distinct characteristics in their profiles: spiky or flat profiles. For the spiky profile applications, there are only a few very hot or scorchingly hot methods, and therefore fully optimizing those methods is sufficient to obtain the best possible performance. The remaining methods mostly have nothing to do with the total system performance. It is relatively easy for any type of profiling mechanism to detect those important methods, and thus this type of application is amenable in general for a dynamic optimization system.

On the other hand, the flat profile applications, such as jack and javac in SPECjvm98, are considered more difficult and challenging for the dynamic compilation system to improve performance. For example, method inlining is less effective for these applications involving a large number of methods that are almost equally important. Consequently, method inlining based on static heuristics, applying the same strategies for any type of method, does not work well for this type of application, as we can see from the relative performance difference between tiny-only and static-inline in Figure 3.

Profile-based inlining is expected to work more effectively than the static heuristics by dynamically changing the inlining decisions depending on the type of application. Since the cost/benefit trade-off line is considered to be higher for flat profile applications than for those with spiky profiles, due to the smaller impact on total performance from inlining individual call sites, method inlining needs to be performed even more selectively. In our implementation, we detect a flat profile for a given group of hot methods based on the average hotness count and the differences between maximum and minimum hotness counts among those hot methods not yet compiled with the highest optimization level. When making a inlining request, we require each edge of the call path to have either a single caller or a very high frequency with a few dominant call sites.

4. EXPERIMENTAL RESULTS

This section presents some experimental results showing the effectiveness and advantages of the profile-directed method inlining in our dynamic compilation system. We outline our experimental methodology first, describe the variations of inlining policies used for the evaluation, and then present and discuss our measurement results.

4.1 Benchmarking Methodology

All the performance results presented in this section were obtained on an IBM IntelliStation M Pro 6849 (Pentium4 2.0 GHz uni-processor with 512 MB memory), running Windows 2000 SP2, and using the JVM of the IBM Developer Kit for Windows, Java Technology Edition, version 1.3.1 prototype build. The benchmarks we chose for evaluating our system are SPECjvm98 and SPECjbb2000 [22]. SPECjvm98 was run in test mode with the default large input size, and in autorun mode with 10 executions for each test, with the initial and maximum heap size of 96 M. SPECjbb2000 was run in the fully compliant mode with 1 to 8 warehouses, with the initial and maximum heap size of 256 M.

The following parameters were used in these measurements. The threshold in the mixed mode interpreter to initiate level-0 compilation was set to 500. The timer interval for the sampling profiler for detecting hot methods was 5 milliseconds. The list of hot methods was examined every 200 sampling ticks. The number of samples to be collected for an instrumentation-based call site profile was set to 1024, and the maximum number of data variations recorded was 8. Exception-directed optimization (EDO) [19] was disabled, because EDO drives its own inlining requests based on the exception path profiles and the mixed inlining requests would make the evaluation for our approach more difficult.

Five inlining policies are compared as described below, one with the current static inlining heuristics, and the others using profile-directed inlining with different degrees of aggressiveness or combined with the static heuristics. The baseline of the comparison is with the policy inlining tiny methods only.

1. **Static:** This employs the existing static heuristics for method inlining in level-1 and level-2 compilation. No profile data on call site information is collected, and the recompilations are requested according to the relative hotness from the sampling profiler.
2. **Profile-1:** This disables all the static heuristics for inlining except for tiny methods, which are always inlined, and small methods, which are inlined in level-1 and level-2 compilations. Inlining other than for these small methods is based solely on the profile information. The inlining requests are conservative in that all the methods called through the hot call path must appear in the group of hot methods.
3. **Profile-2:** This is the same as Profile-1, but more aggressive inlining can be requested for callers that are not in the current group of hot methods. In such cases, the inlining is requested for the methods in the call path within the current group, as well as for the topmost method of the call chain outside of the group.
4. **Hybrid:** This is the same as Profile-1, but also using the current static heuristics if the inlining budget is still not used up after accepting all of the profile-based inlining requests.
5. **Offline:** This is the same as Profile-1, but inlining decisions are made using the offline collected profile information. Section 4.5 gives details regarding how the offline profile information is collected and then used for measurement.

When measuring the compilation overhead in Section 4.3, we measure only level-1 and level-2 compilation statistics, since level-0 is common among all four cases. The compiler is instrumented with several hooks for each part of the optimization process to record the value of the processor's time stamp counter. The priority of the compile thread (for level-1 and level-2) is set higher than normal application threads, so the difference between each value from the time stamp counter is guaranteed to provide time spent for performing the corresponding optimization work, and also the difference in the value between the beginning and the end of the compilation should be the total compilation time.

4.2 Instrumentation and Inlining Statistics

Table 1 shows the compilation statistics when running the benchmarks with the profile-1 inlining policy. The second column of the table shows the total number of methods executed (without the effects of inlining). Native methods are not included. The next five columns show the number of compiled and instrumented methods for each of the optimization levels. No instrumentation is applied to level-2 compiled code, as mentioned in Section 3.1. The count is cumulative, so if a method is level-0 compiled and then promoted to level-1, then it is counted in both categories. The last five columns show the total number of inlining requests made through four steps described in Section 3.2. The number is per call site, so if two inline requests are issued for a method, inlining it into two different call sites within a method, then both are counted as inline requests.

The number of instrumented methods is mostly 10% or less of the compiled methods in level-0 for spiky profile benchmarks (top five in the table) and about 15% to

Benchmarks	# methods executed	level-0		level-1		level-2	inline requests				
		compiled	inst'd	compiled	inst'd	compiled	depth 1	depth 2	depth 3	depth>4	total
227_mtrt	333	182	18	16	4	7	13	2	1	--	16
202_jess	611	311	34	18	11	12	9	6	3	3	21
201_compress	204	118	5	1	0	3	3	--	--	--	3
209_db	199	99	6	2	1	3	6	--	--	--	6
222_mpegaudio	364	207	29	26	10	18	9	2	--	2	13
228_jack	440	363	52	43	15	15	18	4	1	1	24
213_javac	958	823	117	128	24	22	16	3	1	--	20
SPECjbb2000	2663	554	130	129	33	32	18	3	2	2	25

Table 1. Statistics of compiled and instrumented methods for each optimization level, and the number of inlining requests when running benchmarks with the profile-1 inlining policy. The depth columns represent the numbers of call chains for each call path that was requested to be inlined.

20% for flat profile benchmarks (bottom three). The number is about 30% or greater of the compiled methods in level-1. These figures are almost the same as the proportion of methods that are promoted from each level.

The inlining decision seems to work very selectively for each benchmark. The total number of inlining requests relative to the number of compiled methods at level-1 and level-2 is significantly smaller for flat profile benchmarks compared to spiky profile ones, reflecting the strict requirements for call paths to be requested for inlining in terms of the degree of bias in call site distributions.

4.3 Compilation Overhead

Figures 6 to 8 show various data for compilation overhead: the compilation time, compiled code size expansion ratio over the bytecode size, and compilation time peak work memory usage. Smaller bars mean better scores in these figures. The compiled code size used for profile-based inlining policies (profile-1, profile-2, and hybrid) includes additional code space that is dynamically allocated for instrumentation. The peak memory usage is the maximum amount of memory allocated for compiling methods. Since our memory management routine allocates and frees memory in 1 Mbyte blocks, this large granularity masks the minor differences in memory usage and this causes the result of Figure 8 to form clusters.

Overall the profile-based inlining (both profile-1 and profile-2) shows significant advantages over the existing static-heuristics-based inlining in most of the benchmarks. At worst, it does not exceed twice the baseline (the tiny-only inlining policy) in all three metrics of the compilation overhead. In the best case, it is almost equal to the level of tiny-only, apparently due to the minimum number of inlining requests made only for those call sites deemed beneficial for performance improvement. The code size expansion ratio is greatly reduced for some benchmarks, even taking the instrumentation code into account. On average, the compilation overhead (in all three metrics) is just about a 20% to 30% increase from the baseline, and almost a 40% reduction compared to static-heuristics-based inlining.

There are a few exceptions to these general observations. Compress shows degradation with profile-based inlining, and mtrt shows no significant improvement in all three indications of compilation overhead. As shown in Table 1, however, compress has a very few hot methods, resulting in relatively low level of compilation overhead regardless of the inlining policy. In the profile-based inlining case, two inlining requests were actually made for the same method, Compressor/compress, but with different timings because of the group of hot methods supplied by the sampling profiler. Thus the method was compiled twice with full optimization, first with one inlining request and then with an additional request identified later. Because of the low level of compilation overhead, this additional level-2 compilation was enough to cause a seemingly big difference in these ratios. Mtrt suffers for almost the same reason. An additional inlining request was made to one of the hottest methods OctNode/Intersect, after the callee for the call edge had become hot, triggering the second level-2 compilation, and this made the total compilation time increase significantly.

As expected, the hybrid inlining policy generally shows a similar amount of overhead as the static heuristics, but it sometimes causes significantly higher overhead. The overhead for compress is high for the same reason described above, but for the other benchmarks, the poor scores are due to the aggressiveness of the static heuris-

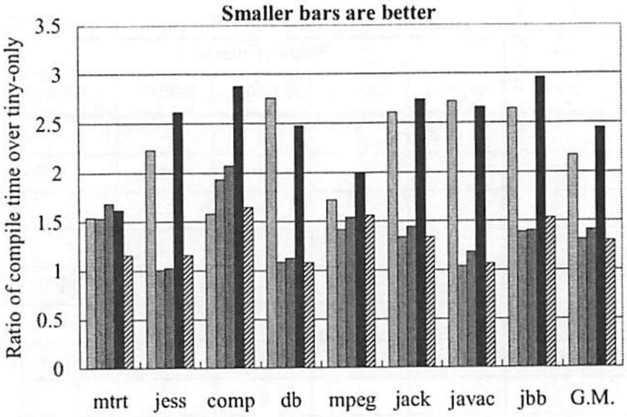


Figure 6. Ratio of compilation time for five inlining policies over the tiny-only case.

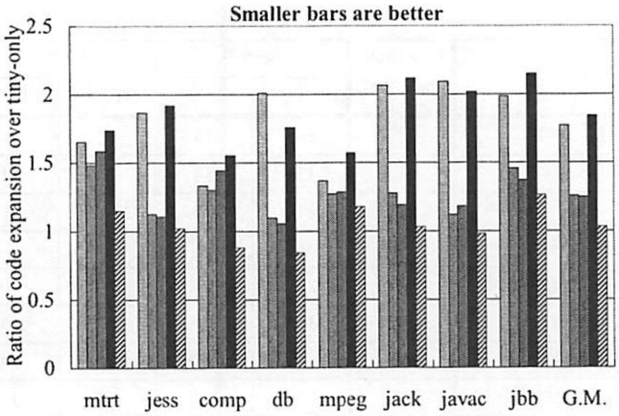


Figure 7. Ratio of compiled code size expansion for five inlining policies over the tiny-only case.

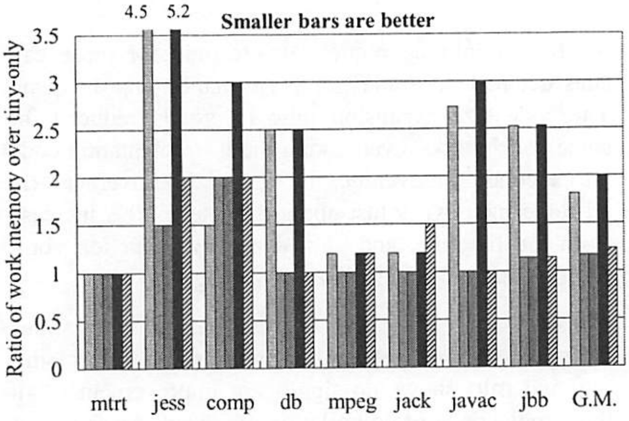


Figure 8. Ratio of compilation time peak memory usage for five inlining policies over the tiny-only case.

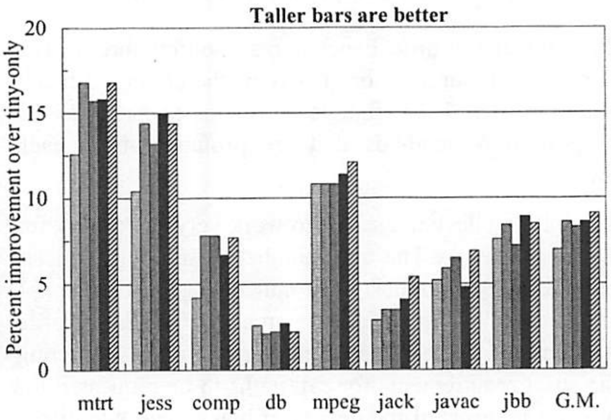


Figure 9. Performance comparison of five inlining policies over the tiny-only case.

▤ Static ▤ Profile-1 ▤ Profile-2 ■ Hybrid ▨ Offline

tics applied after processing the profile-requested inlining. This inlining policy seems unattractive from the viewpoint of compilation overhead.

4.4 Performance Comparison

Figure 9 shows the performance improvement with the five inlining policies over the tiny-only inlining case. Taller bars show better performance. For each inlining policy, we took the best time from 10 executions of an autorun sequence for each test in SPECjvm98 and the best throughput from a series of successive executions from 1 to 8 warehouses for SPECjbb2000. These results show that the profile-based inlining, both profile-1 and profile-2, performs about as well as or better than the static-heuristics-based inlining for most of the benchmarks. The use of profiling seems particularly effective for jess and compress.

Based on the comparison of the actual inlining results between profile-based and static-heuristics-based inlining policies, it was found that for jess, many of the call paths that were identified as important by the profile-based policies were also inlined with the static heuristics, but there were several critical call paths that were not inlined with the static heuristics, and this caused the performance differences. One probable reason for this result is that the chance to inline performance critical call paths is decreased with the static heuristics because the limited inlining budget is wasted by greedily inlining methods based on static assumptions. For compress, the profile-based inlining policy made two critical inlining requests that the static heuristics declined to perform due to the excessive code size for inlining.

The performance is slightly degraded with the profile-based inlining for db. The fact that the hybrid inlining policy restores the performance shows that the current profile-based inlining is missing some opportunities for inlining performance-sensitive call sites in this benchmark. Since the degradation can be observed with the offline-based-profiling as well, it is probably not due to the short interval in online instrumentation to find dominant call edges. Rather it seems that target methods that should have been inlined were overlooked by the sampling profiler as recompilation candidates. The problem may be resolved by expanding the small method criteria, however, this would go against the goal of our approach of basically relying only on profile information. We need to come up with an effective way to deal with this situation.

The profile-2 inlining policy does not seem to produce any additional performance improvement over profile-1 in these benchmarks, regardless of the more aggressive inlining requests.

4.5 Accuracy of Inlining Decision

Because our profile-directed method inlining is on the basis of the call site profiling information collected for a fixed short time interval, there may be a concern as to whether the profile information is accurate enough for driving inlining when compared to a complete profile. In this section we evaluate the accuracy of our profile-based inlining decisions.

We define the accuracy of the inlining decisions relative to the best possible performance improvement and the smallest possible compilation overhead. To obtain complete profiling information during the entire period of program execution, we generated a hook to call a runtime routine at the entry point of the compiled code for each method, taking the caller's address and updating the profile table every time. The total invocation count was also recorded. The information was then written to disk at the end of the run. This provides the equivalent kinds of information to what is collected with our instrumenting profiler at runtime: call site distribution and execution frequency. In the next run, we used the complete profile information as input to make the inlining decisions, without using the instrumentation mechanism. This should provide the upper bound in terms of both performance and compilation overhead improvement for the system with profile-directed method inlining.

Figures 6 to 9 also show how the inlining based on offline collected profile information works relative to other inlining policies. Overall the offline-profile-based inlining finds more opportunities for inlining important call paths and hence is more aggressive compared to the online-profile-based inlining policies. Nevertheless, it shows lower compilation overhead than the profile-1 inlining policy in most of the cases because of the reasons described below.

Compress and mtrt no longer suffer from the problem of additional level-2 recompilation as we saw in Section 4.3. The code size expansion is consistently improved since no instrumentation overhead is imposed. Also some methods can be promoted from level-0 directly to level-2 compilation, because the important call paths for inlining and their impacts are known from the beginning, and this can reduce the total compilation time and code size expansion. All of these are, however, limitations of online-based profiling and its feedback system, and therefore the difference in compilation overhead is considered reasonable.

As for performance, the offline-profile based inlining shows slightly better numbers over profile-1 for some benchmarks, but the difference is not very significant on average. Despite the limited time interval for collecting profile information, our online profiling system seems to capture the majority of the performance sensitive call paths quite well, and hence the method inlining decisions based on this profile information are regarded as effective from the performance point of view.

5. DISCUSSION

We have described the design and implementation of online-profile-directed method inlining in our dynamic compilation framework. We present some observations and discussions based on the results in this section.

Overall the study shows the advantages of the pure profile-based inlining policy (without adding in static heuristics). It shows the potential for either significantly reducing the compilation overhead (measured in time, work memory, and code size) or for improving performance, or both, compared to other inlining policies. In the dynamic compilation environment, we have to be very careful about performing any optimizations that can have significant impact on compilation overhead. The online profiling can provide useful information to guide inlining only for those call sites and call paths that can be expected to produce performance improvements.

As described earlier, our ultimate goal is automatic optimization selections in the dynamic compilation system, and this study shows encouraging results as a first step, since inlining exerts the most influence on both performance and compilation overhead. When inlining decisions are profile-based and the compilation boundaries can be dynamically determined, then the next step will be to estimate the impact of each costly optimization based on the structure of a given method, using indications such as loop structure, characteristics of field and array accesses, and others, as suggested in [2].

The problem described in Section 4.3 is considered inherent to an online profiling and feedback system, since the information is partial, not complete, at any point in time, and we do not know when is the best time to drive recompilation with limited available information. When additional information that can contribute to the performance is later available, we then have to decide whether it is better to trigger additional recompilations. Currently we drive recompilation considering the estimated performance benefit of the additional inlining request based on the relative hotness counts of the caller and the callee, but also limit the maximum number of level-2 compilations for the same method in order to avoid code explosion. We need to explore better ways to manage this situation by, for example, introducing additional metrics of the impact of inlining on a particular call path.

In this paper, we consider only the relative strengths and distributions of the call edges when driving inlining. However, the significant impact of method inlining is not only through the direct effect of eliminating call overhead, but also due to indirect effects of specializing an inlined method body into the calling context, with better utilization of dataflow information at the call sites. Since our instrumentation mechanism can collect parameter values as well as return addresses, it should be possible to estimate whether inlining a method will be beneficial in terms of the effect of these optimizations as well, based on the result of impact analysis [23]. We will try to exploit this information in the future for pursuing better inlining strategies.

6. RELATED WORK

There are several previous studies that evaluate method inlining using profile information, mostly collected offline, in both static and dynamic compilation systems. Overall, it was concluded that the use of profile information for inlining is effective to improve program performance under careful management of the increase of the static code size.

Scheifler [21] shows that inlining optimization can be reduced to the well-known knapsack problem. He uses a greedy algorithm to minimize the estimated number of function calls subject to a size constraint. This relies on the runtime statistics of the program, collected with various sets of input data, to calculate the expected overhead of each invocation. The constant ratio assumption is used to avoid the cost of a multi-level history in gathering statistics. Kaser and Ramakrishnan [16] propose a probabilistic model to estimate the effect of using profile data, with a one-level history based on the constant ratio assumption. When evaluating inlinable calls remaining after optimization, they report good results with their technique compared to other compilers.

Chang et al. [7] describe profile-guided procedure inlining with the IMPACT optimizing C compiler. They first construct a weighted call graph using the offline-collected profile information on the number of invocations of each function and relative hotness counts of each call edge. A greedy algorithm is then applied bottom-up in the call graph to maximize the number of reductions of dynamic function calls while keeping the code size expansion within a fixed bound. Their result shows a significant performance improvement with a relatively low code expansion ratio. However they don't report on how much of the effectiveness is contributed by the use of profile information. Dynamically dispatched calls (through function pointers in C programs) are not inlined in this study.

Ayers et al. [5] describe the design and implementation of the inlining and cloning in the HP-UX optimizing compiler, and show the performance of the SPECint92 and 95 benchmarks can be substantially improved with their technique. They use profile information collected offline to prioritize the inlining or cloning candidates of the call site to be considered. The use of profile information is reported to be quite effective, however it is an intra-procedural profile of the basic block level execution frequency, not information on call edges for guiding inlining for a particular call path.

Arnold et al. [4] present a comparative study of static and profile-based heuristics for inlining with several limits on code expansion. In considering three inlining heuristics, based on a static call graph, a call graph with node weights, and a dynamic call graph with edge weights, they regard the selection of inlining candidates as a knapsack problem, and employ a greedy heuristic based on the benefit/cost ratio as a meta-algorithm for approximating the NP-hard problem. Their experiment, done with offline based profiling and an ahead-of-time compilation framework, shows that a substantial (sometimes more than 50%) performance improvement can be obtained with the heuristics based on the dynamic call graph with edge weights over the static call graph, even with modest limits on code size expansion. This work became the basis for the feedback-directed method inlining in adaptive optimization implemented in the Jalapeño JVM (now called the Jikes Research Virtual Machine) [1]. This system periodically takes a statistical sample of the method calls and maintains an approximation to the dynamic call graph during the course of the entire program execution. The online-profile-directed method inlining is shown to contribute to significant performance improvement for some benchmarks. Arnold [3] further continues to improve the effectiveness of the feedback-directed inlining by collecting execution frequencies of the control flow edges between basic blocks and letting this information be used for fine-tuning the inlining decisions.

Dean and Chambers [10] describe a system based on the SELF-91 compiler that uses the first compilation as a tentative experiment and records inlining decisions and the benefits of the resulting effect on optimizations in a database. The compiler can then take advantage of the recorded information for future inlining decisions by searching the database with the known information about the receiver and arguments. They do not exploit runtime profile information in their system, though the expected execution frequency of the call site is used for inlining decisions.

The SELF-93 system [14] is an adaptive recompilation system using online profile information. It collects call-site-specific profile information for receiver class distributions (type feedback) in un-optimized runs, and then when the method is recompiled, makes use of this information to optimize dynamically-dispatched calls by predicting likely receiver types and inline calls for these types. It was demonstrated that the performance of many programs written in SELF can be substantially improved with this technique. Grove et al. [12] complement this result by studying the various characteristics of the profiles of receiver class distributions collected offline, such as the degree of bias, effectiveness of deeper granularity, the stability across input and programs. They report that the compiler can effectively use deeper granularity of profile context to predict more precisely the target of dynamically dispatched procedure calls.

HotSpot [20] is a JVM product implementing an adaptive optimization system with an interpreter to allow a mixed execution environment. As in the SELF-93 system, it also collects profiles of receiver type distribution online in the interpreted execution mode. The profile information, together with class hierarchy analysis, is used when optimizing the code for virtual and interface calls.

There are several systems that use annotations to specify dynamic optimizations. Krintz and Calder [17] describe an annotation framework for reducing compilation overhead for Java programs. One of the proposed annotations is method inlining on the basis of analysis of profile information collected offline, which allows substantial reduction of startup overhead. Mock et al. [18] present Calpa, a system to automatically generate annotations for the DyC dynamic compiler. It evaluates the offline-collected information regarding basic block execution frequencies and a value profile based on its own cost/benefit model, and determines runtime constants for specialization and dynamic compilation strategies.

7. CONCLUSIONS

We have described an empirical study of online-profile-directed method inlining in our Java dynamic compilation system. We removed the existing static heuristics of method inlining except when dealing with tiny and small size methods, and rely solely on the online profile information to drive the method inlining in level-1 and level-2 optimizations. We presented our design and implementation showing how the information on call site distribution and execution frequency can be collected at runtime, and then showed how the hot call paths can be extracted to identify potential candidates for method inlining. We evaluated our approach using the industry standard benchmarks, and showed its potential to achieve better performance and/or smaller compilation overhead (measured in compilation time, peak memory usage, and code size expansion ratio) when compared to the static inlining heuristics.

In the future, we will further study the effectiveness of profile-directed method inlining by examining the accuracy and efficiency of our inlining approach when compared to other approaches such as dynamic call graph construction at sampling time [1].

8. ACKNOWLEDGMENTS

We would like to thank all the members of the Network Computing Platform group in IBM Tokyo Research Laboratory for helpful discussions and comments. The anonymous reviewers also provided many valuable suggestions and comments on the presentation of the paper.

REFERENCES and NOTES

[1] M. Arnold, S. Fink, D. Grove, M. Hind, and P.F. Sweeney. Adaptive Optimizations in the Jalapeño JVM. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications*, pp. 47-65, Oct. 2000.

[2] M. Arnold, S. Fink, D. Grove, M. Hind, and P.F. Sweeney. Adaptive Optimizations in the Jalapeño JVM: The Controller's Analytical Model. In *Proceedings of the ACM SIGPLAN Workshop on Feedback-Directed and Dynamic Optimization, FDDO-3*, Dec. 2000.

[3] M. Arnold. Online Instrumentation and Feedback-Directed Optimization of Java. Technical Report DCS-TR-469, Department of Computer Science, Rutgers University, Nov. 2001.

[4] M. Arnold, S. Fink, V. Sarkar, and P.F. Sweeney. A Comparative Study of Static and Profile-Based Heuristics for Inlining. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, Jan. 2000.

[5] A. Ayers, R. Gottlieb, and R. Schooler. Aggressive Inlining. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 134-145, Jun. 1997.

[6] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 1-12, Jun. 2000.

[7] P.P. Chang, S.A. Mahlke, W.Y. Chen, and W.W. Hwu. Profile-Guided Automatic Inline Expansion for C Programs. *Software Practice and Experience* 22(5), pp. 349-369, May 1992.

[8] W.Y. Chen, P.P. Chang, T.M. Conte, and W.W. Hwu. The Effect of Code Expanding Optimizations on Instruction Cache Design. *IEEE Transactions on Computer* 42(9), pp. 1045-1057, Sep. 1993.

[9] M. Cierniak, G.Y. Lueh, and J.M. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 13-26, Jun. 2000.

[10] J. Dean, and C. Chambers. Training Compilers for Better Inlining Decisions. Technical Report 93-05-05, Department of Computer Science and Engineering, University of Washington, May 1993.

[11] D. Detlefs, and O. Agesen. Inlining of Virtual Methods. In *The 13th European Conference on Object-Oriented Programming, ECOOP*, Lecture Note in Computer Science, 1628, pp. 258-277, 1999.

[12] D. Grove, J. Dean, C. Garrett, and C. Chambers. Profile-Guided Receiver Class Prediction. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications*, pp. 108-123, Oct. 1995.

[13] R.E. Hank, W.W. Hwu, and B.R. Rau. Region-Based Compilation: An Introduction and Motivation. In *Proceedings of 28th International Conference on Microarchitecture*, pp. 158-168, Dec. 1995.

[14] U. Hölzle. Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming. Ph.D. Thesis, Stanford University, CS-TR-94-1520, Aug. 1994.

[15] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. A Study of Devirtualization Techniques for a Java Just-In-Time Compiler. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications*, pp. 294-310, Oct. 2000.

[16] O. Kaser, and C.R. Ramakrishman. Evaluating Inlining Techniques. *Computer Languages*, 24(2) pp. 55-72, 1998.

[17] C. Krintz, and B. Calder. Using Annotations to Reduce Dynamic Optimization Time. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 156-167, Jun. 2001.

[18] M. Mock, C. Chambers, and S. Eggers. Calpa: A Tool for Automating Selective Dynamic Compilation. In *Proceedings of 33rd International Conference on Microarchitecture*, pp. 1-12, Dec. 2000.

[19] T. Ogasawara, H. Komatsu, and T. Nakatani. A Study of Exceptions and its Dynamic Optimization for Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications*, pp. 83-95, Oct. 2001.

[20] M. Paleczny, C. Vick, and C. Click. The Java HotSpot Server Compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM '01)*, pp. 1-12, Apr. 2001.

[21] R.W. Schiefler. An Analysis of Inline Substitution for a Structured Programming Language. *Communications of the ACM*, 20(9), pp. 647-654, Sep. 1977.

[22] Standard Performance Evaluation Corporation. SPECjvm98 and SPECjbb2000 benchmarks available at http://www.spec.org/osg.

[23] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A Dynamic Optimization Framework for a Java Just-In-Time Compiler. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications*, pp. 180-194, Oct. 2001.

[24] J. Whaley. A Portable Sampling-Based Profiler for Java Virtual Machines. In *Proceedings of the ACM SIGPLAN Java Grande Conference*, pp. 78-87, Jun. 2000.

[1] Note that each compilation level does not match the corresponding level of optimization described in [23], although the number of optimization levels is the same. The elements within each level have been rearranged. The purpose of level-0 compilation is to allow transition from MMI to compiled code with a lower threshold value without causing additional compilation overhead, and to provide the system with profile information for a broader range of methods.

[2] This estimate excludes the prologue and epilogue code. The compiled code size is estimated based on the sequence of bytecodes each of which is assigned an approximate number of instructions generated.

[3] The instrumentation code is just a small piece of code, less than 100 instruction bytes, and the corresponding table space is in practice less than another 100 bytes. This space is treated in the same way as the compiled code so that it can be reclaimed upon class unloading.

[4] Since tiny methods are always inlined in the compiled code and thus the call site found in the profile may actually be within an inlined tiny method, the runtime structure, IMF, is consulted to get the exact call path from the call edge.

Stress-testing Control Structures for Dynamic Dispatch in Java

Olivier Zendra
INRIA Lorraine - LORIA / McGill University
615 Rue du Jardin Botanique, BP 101
54602 Villers-Les-Nancy Cedex, France
`http://www.loria.fr/~zendra`
`Olivier.Zendra@loria.fr`

Karel Driesen
McGill University
School of Computer Science - ACL Group
Montreal, Quebec, Canada H3A 2A7
`http://www.cs.mcgill.ca/~karel`
`karel@cs.mcgill.ca`

ABSTRACT

Dynamic dispatch, or late binding of function calls, is a salient feature of object-oriented programming languages like C++ and Java. It can be costly on deeply pipelined processors, because dynamic calls translate to hard-to-predict indirect branch instructions, which are prone to causing pipeline bubbles. Several alternative implementation techniques have been designed in the past in order to perform dynamic dispatch without relying on these expensive branch instructions. Unfortunately it is difficult to compare the performance of these competing techniques, and the issue of which technique is best under what conditions still has no clear answer. In this study we aim to answer this question, by measuring the performance of four alternative *control structures* for dynamic dispatch on several *execution environments*, under a variety of precisely controlled execution conditions. We stress test these control structures using micro-benchmarks, emphasizing their strenghts and weaknesses, in order to determine the precise execution circumstances under which a particular technique performs best.

Keywords

Java, dynamic dispatch, control structure, optimization, JVM, binary tree dispatch, virtual function call

1. INTRODUCTION

Object-oriented message dispatch is a language concept that enables data (objects) to provide a functionality (message) by relying on a type-specific implementation, or method. At run time, the object that receives a message, or virtual method call, retrieves the corresponding class-specific method and invokes it. This late binding of dispatch targets allows any object to play the role of the receiver object, as long as the new object implements the expected interface (is substitutable à la Liskov [Lis88]). Such type-substitutability enables better code abstraction and higher code re-use, and is therefore one of the main advantages of object-oriented languages.

As a consequence, dynamic dispatch occurs frequently in object-oriented programs. For instance, virtual method invocations in Java [GJSB00] occur every 12 to 40 byte codes [DLM$^+$00] in SPEC JVM98. Such late-bound calls are typically expensive on modern deeply pipelined processors, because they translate to hard-to-predict indirect branch instructions that are a cause for long pipeline bubbles [DHV95].

It is nonetheless exceedingly difficult to precisely measure the time spent on dynamic dispatch itself by real object-oriented programs. Indeed, virtual function calls occur frequently, which makes it difficult to isolate dispatch time from the runtime of regular code. Furthermore, call frequency and amount of runtime polymorphism strongly depend on coding style as well as runtime parameters. Finally, on modern superscalar processors, call code sequences can be co-scheduled with regular code, which further blurs the picture. Dispatch overhead therefore depends not only on the dispatch code sequence itself, but also on the code surrounding the call and on the processor ability to detect and take advantage of instruction level parallelism.

An estimate of dispatch overhead, based on real programs and relying on super scalar processor simulation, can be found in [DH96]. The authors measure a median dispatch overhead of 5.2% in C++ programs and 13.7% in C++ programs with all member functions declared virtual (as is the default in Java). For one program, the overhead was as high as 47% of the total execution time. While evidence from practice suggests that most Java programs exhibit little polymorphism at run time, it is true that for some programs the optimizations tested in this study can make as much as a 50% difference in execution time, as demonstrated by the micro-benchmarks. It thus appears very sensitive to optimize dynamic dispatch, in order to avoid incurring a significant performance penalty when relying on the object-oriented design style.

Alternative implementation techniques are available to perform dispatch to multiple targets without using expensive branch instructions. Unfortunately, comparing the performance of these competitive techniques is hard, and the literature typically reports measurements of few alternatives, on only one execution environment.

In this study, we propose and report on the results of a proof-of-concept methodology to measure the performance of several control structures for dynamic dispatch on a *variety of Java Virtual Machines and hardware platforms.* In this first-step study, we rely on micro-kernel benchmarking to determine and magnify the relative performance of control instructions under a large number of *varying execution conditions.*

The results show, among other things, that:

- Virtual method call performance is highly *dependent on the execution pattern* at a particular call site
- When the call site has a low (2-3 target types) to medium (6-8 target types) degree of polymorphism, optimizations are possible that *improve performance across JVMs and hardware platforms* (that is, platform independent optimization)
- *Processor architecture shines through,* especially on high-performance JVMs: the performance profile from different VMs executing on the same hardware look similar, those from the same VM executing on different hardware look different.

This paper is organized as follows. Section 2 reviews dynamic dispatch implementations and related work at software, run-time system and hardware level. Section 3 presents our methodology and the experimental setup. Section 4 presents some of our results and discusses them. Finally, section 5 concludes and points at future research directions.

2. BACKGROUND

2.1 Monomorphism vs. Polymorphism

Dynamic dispatch is expensive because the target method depends on the run-time type of the receiver, which generally cannot be determined until actual execution.

Many different optimization techniques have thus been proposed, which can be seen as falling into two broad categories:

Optimizing monomorphic calls Since dynamic dispatch is expensive, the fastest way to do it is to avoid it altogether.

Various kinds of *program type analysis* (such as [DGC95, SLCM99, SHR$^+$00]) enable the devirtualization of provably monomorphic calls (calls with only one target type), replacing the expensive late-bound call by by a direct, cheaper, early-bound call. This technique has the added advantage of allowing inlining of target methods, thus stripping away all of the call overhead and enabling a more radical optimization of the inlined code by classical methods.

Dynamic optimization (e.g [DDG$^+$96, HU94]) such as employed by the SUN HotSpot^{TM} Server JVM allows method inlining at run time, which permits further optimization of calls that are monomorphic in only a particular run of the program, even though multiple target types are possible after compile time.

Optimizing polymorphic calls In spite of all efforts, some calls cannot be resolved as monomorphic. Optimizing the remaining polymorphic ones (calls with more than one target type) is crucial.

Program type analysis can also optimize these polymorphic calls, especially when the number of possible types is very low. For example, a compiler can replace a late-bound call with two possible target types by a conditional branch and two static, direct, early-bound calls. At run time, a cheap conditional branch and cheap static call are executed instead of one expensive late-bound call (strength reduction). Such a strength reduction operation is usually a win on current processors, since prediction of conditional branches is easier than prediction of indirect branches. Furthermore, the dominant (most common) call direction can be inlined, leading to similar optimization opportunities as for monomorphic calls, with the guard of a cheap conditional branch [AH96].

Dynamic optimization can also replace a call that is dominated by one target type at run time, enabling the same operation as above with increased type precision.

These solutions to optimize dynamically dispatched calls are amenable to two approaches: hardware and software.

2.2 Hardware Solutions

Virtual method invocations in Java translate, in the native machine code, into two dependent loads followed by an *indirect branch* (or indirect jump). This indirect branch is responsible for most of the call overhead [Dri01]. Branches are expensive on modern, deeply pipelined processors because the next instruction cannot be fetched with certainty until the branch is resolved, typically at a late stage in the pipeline (e.g., after 10-20 cycles on a Pentium III).

Most processors try to avoid these pipeline bubbles by speculatively executing instructions of the most likely execution path, as predicted by separate *branch prediction* micro-architectures. For example, a Branch Target Buffer (BTB) stores one target for each indirect, multi-way branch and can predict monomorphic branches with close to 100% accuracy, which removes the branch misprediction overhead in the processor.

Unfortunately, polymorphic calls are harder to predict. Sophisticated two-level indirect branch hardware predictors [CHP98] can provide a similar advantage as a BTB for multi-target indirect branches that are "regular" and whose target correlates with the past history of executed branches.

Unfortunately, indirect branches are more difficult to predict than conditional branches. A conditional branch has only one target, encoded in the instruction itself as an offset, so a processor only needs to predict whether the conditional branch is taken or not (one bit). Indirect branches can have many different targets and therefore require prediction of the complete target address (32 or 64 bits). Sophisticated predictors [DH98a, DH98b] can reach high prediction rates, but generally require large on-chip structures. Indirect branch predictors thus tend to be more costly and in practice less accurate than conditional branch predictors (Branch History Buffers, BHTs), even in modern processors.

Therefore, replacing at the code level a rather unpredictable indirect, multi-way branch by one or several more predictable conditional branches followed by a static call seems a likely optimization, helping the processor. This strength reduction of control structures is exploited by several of the techniques in the next section.

2.3 Software Solutions

Most JVMs include some way to de-virtualize method invocations that are actually monomorphic, by replacing the costly polymorphic call sequence by a direct jump. For example, various forms of whole program analysis (e.g., [BS96, SHR$^+$00]) show that most invocations in object-oriented languages are monomorphic.

Some JVMs use a dynamic approach. For example, HotSpot relies on a form of inline caching [DS84, UP87]. The first time a virtual method invocation is executed, it is replaced by a direct call preceded by a type check. Subsequent executions with the same target are thus direct, whereas executions with a different target fall back to a standard virtual function call.

Actual run-time polymorphism can also be optimized in software, for example by using Binary Tree Dispatch (BTD), as implemented in the SmallEiffel compiler [ZCC97]. BTD replaces a sequence of powerful dispatch instructions using an indirect branch by a sequence of simpler instructions (conditional branches and direct calls). When the sequence of simple instructions remains small, it can be more efficient than a call through a virtual function table, and should perform particularly well on processors with accurate conditional branch prediction and large BHT. A BTD is a static version of what is commonly known as a Polymorphic Inline Cache [HCU91]. A PIC collects targets dynamically at run time (it is a restricted form of self-modifying code), effectively translating a lengthy method lookup process into a sequential search through a small number of targets. The *if sequence* control structure exercised in our micro benchmark suite (see section 3.3) is akin to the implementation of a PIC described in [HCU91]. As in the latter paper, we found that megamorphic [DHV95] call sites (more than 10 possible target types) are too large for a sequential *if* to be cost-effective.

Chambers and Chen also proposed a hybrid implementation mechanism [CC99] for dynamic dispatch that can choose between alternative implementations of virtual calls based on various heuristics. The experiments in our study complement their approach, since we aim to more precisely define the gains and cutoff points reachable with each technique on multiple platforms.

3. METHODOLOGY

3.1 Overview

We started this work in order to find out whether control structure strength reduction could be used to optimize dynamic dispatch under specific execution conditions and across different hardware platforms, i.e. to find out whether platform independent optimization is feasible.

In order to allow platform independent optimization to be effective, two conditions must hold. First, strength-reducing operations must be guided by platform independent information; the analysis may include profile data if it is not platform-specific. Second, the performance of control structures must be consistent across platforms.

The first condition is fulfilled by various forms of static program analysis and program-level profiling, and many studies show that optimizable call sites are common.

The second condition needs to be verified. Even the reasonable assumption that direct static calls are faster than monomorphic virtual ones may not always hold in practice due to implementation features, at the virtual machine level or at the processor micro-architecture level. For instance, a Pentium III stores the most recent target of indirect branches, which can make monomorphic virtual calls as efficient as static calls.

In the next section we discuss our experimental framework to measure performance of control structures across different JVM and hardware platforms.

3.2 Experimental Setup

Since we focus on polymorphic calls, a large variety of execution behaviors and control structures has to be measured on several platforms. Therefore, we design a comprehensive suite of Java micro benchmarks as a proof-of-concept simulation of various implentations of dynamic dispatch in various JVMs. This allows us to test the performance of control structures under controlled execution conditions, leveraging the wide availability of the Java VM to measure on different execution environments.

All benchmarks use the same superstructure: a long-running loop that calls a static routine which performs the measured dispatch. The receiver object (actually, its type ID) is retrieved from a large array, which is initialized from a file that stores a particular execution pattern as a sequence of type IDs. This initialization process ensures that compile-time prediction of the type pattern is impossible. Different files store a variety of type ID sequences, representing different patterns and degrees of polymorphism.

The experimental parameter space thus varies along three dimensions:

Control structures How do different different control structures for dynamic dispatch perform?

Execution patterns This dimension has three related sub-dimensions. First, the *static number* of possible receiver types at the dispatch site, which influences the program code and can be determined by program analysis before execution. Second, the *dynamic number* of receiver types at the dispatch site, that is the range of types occurring in a particular program run. Third, the *pattern* of receiver type IDs, that is the order and variability of receiver types at run time.

Execution environments This dimension has two related sub-dimensions: the *virtual machine* used and the *processor* it is run on.

Each data point (timing) within this parameter space is measured as follows. First, the benchmark is run 5 times over a long (10 million) loop, which gives a "long run average" running time. This average comprises only the loop part (not the initialization). When executed on dynamically optimizing JVMs such as HotSpot, this execution time comprises both the execution as "cold code" and the execution as optimized once the optimizer has determined the loop is a "hot" one. The JVM is thus given ample opportunity to fully optimize control structures. Then, the benchmark is re-run 5 times over a very long (60 million) loop, which provides a "very long run average". The difference between these two averages, "long" and "very long", represents only "hot", optimized loops, and gives us our final result, after normalization to 10 million loops.

The three dimensions of the parameter space are detailed in sections 3.3, 3.4 and 3.5.

3.3 Various control structures

We measure a variety of control structures for dynamic dispatch implementation. Although it is not comprehensive, we believe it covers the main possibilities available to optimizing compilers at the bytecode and native code level.

Virtual calls At the Java source code level, a dispatch site is a simple method call: `x.foo()`. At the Java bytecode level, a special instruction, `invokevirtual`, is provided to implement virtual calls. The dynamic dispatch instruction uses the message signature (argument to the `invokevirtual` bytecode) and the dynamic type of the receiver object (atop the stack) to determine the actual target method. Generally, this translates at the hardware level into a table-based indirect call [ES90]. This constitutes the first implementation of dynamic dispatch we tested, in our "Virtual" series of micro-benchmarks.

It is however possible to use other control structures, based on simpler bytecode instructions, such as type equality tests followed by static calls. These control structures can take at least three forms:

If sequence First, a sequence of 2-way conditional type checks can be used. For example, let's assume a polymorphic site `x.foo()` where global, system-wide analysis detected that the receiver could only have four possible concrete types at runtime: T_A, T_B, T_C and T_D. The corresponding pseudo-code is shown in figure 1, where the tests discriminate between all the known possible types and lead to the appropriate leaf static call. This implementation of dynamic dispatch is tested in our "IfSequence" series of micro-benchmarks, where the type ID is an integer stored in an extra field of every object.

```
xTypeID = x.typeID;
if (xTypeID == ID_FOR_TYPE_A) then
    A.static_foo(x);
else if (xTypeID == ID_FOR_TYPE_B) then
    B.static_foo(x);
else if (xTypeID == ID_FOR_TYPE_C) then
    C.static_foo(x);
else if (xTypeID == ID_FOR_TYPE_D) then
    D.static_foo(x);
endif
```

Figure 1: P-code for if-sequence dispatch

A variant of this would not test against a type ID field added to the objects, but use a series of `instanceofs` expressions. This technique would avoid the need for the type ID field, and the associated space and initialization overhead. Its performance compared to the *if sequence* above would mostly be related to the relative performances of the `instanceof` and `getfield` bytecodes instructions. We did not include this variant in our benchmarks.

Another variant consists in accessing the type descriptor (`Class` object) of the receiver, using the `getClass()` method, instead of the type ID field. Since `getClass()` is a final native function of `Object`, with JVM support, it is likely to be quite fast. This technique also avoids the need for the type ID field and the associated costs. However, the type test would have to be done against `CLASS_FOR_TYPE_A`, `CLASS_FOR_TYPE_B`, etc. instead of `ID_FOR_TYPE_A`, `ID_FOR_TYPE_B`, etc. Retrieving each of these class descriptors incurs a cost as well, using either a static method for each class, or (in the context of our proof-of-concept Java micro-benchmarks) the more general function `forName (String className)` in class `Class`, whereas the type IDs are constants. This variant thus also seems to be potentially slower. We did not include it in our benchmarks.

Binary Tree Such 2-way conditional tests can be organized more efficiently, as a binary decision tree [ZCC97]. Let's assume the type IDs corresponding to the types T_A, T_B, T_C and T_D, are, respectively, 19, 12, 27 and 15. Then, the pseudo-code generated for `x.foo()` looks like the one in figure 2. We test this implementation of dynamic dispatch in our "BinaryTree" series of micro-benchmarks.

```
xTypeID = x.typeID;
if (xTypeID <= 15) then
    if (xTypeID <= 12) then
        B.static_foo(x);
    else
        D.static_foo(x);
    endif
else
    if (xTypeID <= 19) then
        A.static_foo(x);
    else
        C.static_foo(x);
    endif
endif
```

Figure 2: P-code for binary tree dispatch

Note that a BTD using the `getClass()` variant described above for *if sequence* would be slower than the one we present, since the tests in the dispatch tree would not be done with constants anymore. We thus did not include this variant in our benchmark suite.

Switch Finally, a multi-way conditional instruction can be used, namely a Java dense `switch`, translated into a `tableswitch` bytecode instruction, whose suggested implementation [LY99] by the JVM is an indirection in a table. The corresponding pseudo-code, tested in our "Switch" series of micro-benchmarks, is shown in figure 3.

For the sake of simplicity, we only test dense switches. Indeed techniques exist (global analysis, coloring,...) to have a compact allocation of type IDs. In case of sparse type IDs, sparse switches should be translated into a standard `lookupswitch` [LY99] bytecode instruction, that can be implemented by the JVM as a series of *ifs* or a binary search, thus falling back to one of the techniques we already present in this study.

```
xTypeID = x.typeID;
switch (xTypeID)
    case ID_FOR_TYPE_A then
        A.static_foo(x);
    case ID_FOR_TYPE_B then
        B.static_foo(x);
    case ID_FOR_TYPE_C then
        C.static_foo(x);
    else ID_FOR_TYPE_D then
        D.static_foo(x);
endswitch
```

Figure 3: P-code for tableswitch dispatch

The general idea behind strength reduction for dynamic dispatch is that simpler instructions, although more numerous, should be more predictable and executed faster than complex instructions.

All these control structures, except the plain `invokevirtual`, have a size that is proportional to the number of tested types. When used to implement dynamic dispatch, without any fall-back technique, all possible types have to be tested; this set of possible types thus has to be determined by a global analysis. This is accounted for in our benchmark suite, by creating, for each distinct dispatch technique, several benchmarks differing only by the number of types they can handle.

In the last three control structures, the leaf calls are purely monomorphic. They are thus implemented as Java static calls `X.static_foo(x)`, with the original receiver object being passed as the first argument (instead of being the implicit `this` argument in the virtual call). We thus gave a "StaticThisarg" suffix to these benchmarks. We also benchmark leaves implemented as monomorphic virtual calls which, as we expected, turn out to be generally slower than the static leaves. As a consequence, we do not detail those results in this paper.

Note that the last three techniques may also be used to serve as run-time adaptive caches catching the most frequent or more recent types, preceding a more general fall-back technique. In this case, they would be akin to PICs, or more accurately, as different alternative control structures which can be used to implement various sizes of PICs.

Figure 4 shows a synthetic comparison of these four control structures.

3.4 Various type patterns

The runtime behavior of the program is another crucial factor in the performance of a given dynamic dispatch site. In order to simulate varying behaviors while keeping precise control, we timed our benchmarks by generating various type ID patterns. Each micro benchmark reads a particular pattern from file at run time to initialize a 10K int array holding type IDs, which is then iterated over a large number of times.

For this study, we used synthetic patterns which represent extremes in program behavior. We plan to use real applications or real application traces in future work. We decided to design patterns comprising between one and 20 possible receiver types, in order to cover a wide range of cases. In most real applications though, the degree of polymorphism remains typically much smaller (3 to 5). The low degrees of polymorphism in our tests thus have a lot of importance for most cases, while higher degrees tend to show how a specific technique scales up. The following four patterns are presented below and in figure 5: the constant pattern, the random pattern, the cyclic pattern and the stepped pattern.

Constant This pattern is the simple 100% monomorphic case, where the receiver type is always the same and is thus perfectly predictable. This is a very common case. Various techniques detect such monomorphic dispatch sites and get rid of them by replacing them with direct calls (de-virtualization). However, these techniques may not always be applied, do not detect all monomorphic call sites and do not handle call sites that are in principle polymorphic but never change targets within any single run. It is thus worth testing the behavior of dynamic dispatch techniques on this best-case constant pattern. Since the value of the constant type ID influences performance, we have to test various IDs within the static range.

Random This pattern is the exact opposite of the previous one: it can't be predicted, features high polymorphism (many receiver types) and high variability (many changes during execution). As such, it represents a worst-case scenario likely to be rare in object-oriented programs.

Cyclic The cyclic pattern features a regular variation of the type ID, each ID being the previous one incremented by 1 up to maxID and back to 1, and so on. This pattern is thus highly polymorphic and has a very high variability (the type changes at every call), like the random pattern, but is still very regular. Advanced micro-architecture such as two-level branch predictors are capable of detecting some

Structure	Pros	Cons
Virtual	Short code sequence. Size independent of # of static types.	Translates to expensive indirect branch. Generally slow.
If Sequence	Uses an inexpensive static call. Fast for types at beginning of sequence. Translates to conditional branches better predicted by hardware than indirect ones.	Long code sequence. Slow for types at end of sequence. Size depends on # of static types.
Binary Tree	Uses an inexpensive static call. Equally fast for all types (distance to leaf). Translates to conditional branches better predicted by hardware than indirect ones.	Long code sequence. Speed depends on # of static types (log2). Size depends on # of static types.
Switch	Uses an inexpensive static call.	Long code sequence. Size depends on # of static types. Unreliable speed: depends on JVM.

Figure 4: Comparison summary of various control structures

cyclic branch behavior and therefore should predict this pattern accurately, especially for small cycles. As such, and even though it is probably fairly uncommon in OO programs, this pattern represents a kind of intermediate point between constant and random.

Stepped This pattern is a regular variation of the cyclic pattern, close to the constant pattern in behavior. It features a variation of the type ID from 1 to maxID, with increments of 1, but with as few changes as possible within a single run. It thus exhibits long, constant steps, whereas the cyclic pattern has a step length of 1. The stepped pattern has the same degree of polymorphism as the cyclic one (same number of types), but much lower variability. It should thus be highly predictable, even by simple predictors such as a Branch Target Buffer. This stepped pattern is probably quite common in object-oriented programs, for example when iterating over containers of objects, which often contain instances of a single type.

3.5 Various execution environments

The execution environment is the last varying dimension in our study and consists of two parts: the hardware platform and the virtual machine used to execute the benchmarks. Running different virtual machines is similar to testing a particular program using different compilers. The addition of an extra execution layer, the JVM, makes execution more complex and makes it significantly harder to interpret performance results, but it provides platform-independence and is thus essential to our approach.

The benchmark suite was run on three hardware platforms:

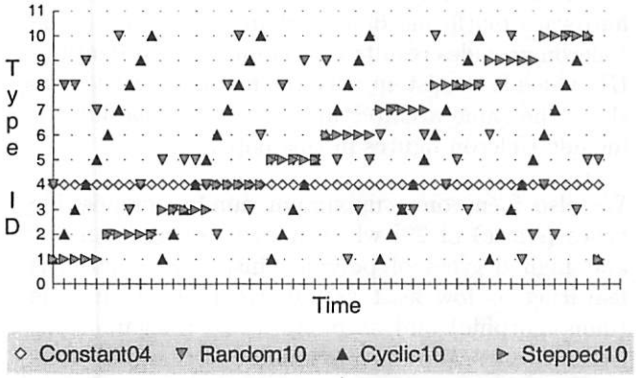


Figure 5: Patterns dynamic behavior

SUN UltraSparc III This machine is based on one 750 MHz processor and 1 GB of RAM, with SunOS 5.8.

Intel Pentium III This machine has dual 733 MHz processors, with 512 MB of RAM, running Linux Mandrake with kernel 2.2.19. Note that for our benchmarks, dual processor capability should have little if no impact.

Intel Celeron This lower-end machine comprises one 466 MHz Celeron with 192 MB of RAM and Linux Mandrake with kernel 2.2.17.

Of course, not all JVMs are available on all hardware platforms. Furthermore, the fact that a JVM is available under the same name on several different OS and hardware platforms is no guarantee at all they are indeed the same JVM: their back-ends for instance must be different. The JVMs tested during this study are generally in their 1.3.1 version. We show the IBM JVM (known as "the Tokyo JIT") and the SUN HotSpot Server as examples of high-performance JVMs and the SUN HotSpot Client, which is the most widely available JVM, and runs on many different hardware platforms.

The following result section shows the essence of the large amount of data gathered.

4. RESULTS AND DISCUSSION

As explained in the previous section, we measure the performance of different control structures in a number of varying dimensions: hardware, JVM, number of possible types (static) and type pattern (dynamic). This leads to a vast parameter space, in which we gather a very large number of data points (more than 21,000).

For space constraints reasons, we cannot show all the data and therefore we pick a representative sample: the dual Pentium III and the UltraSparc III, two hardware platforms described in section 3.5. The Celeron provides results very similar to the Pentium III, which is consistent with the fact both processors share the same architectural core; we thus did not include Celeron figures in this paper.

We also focus on a maximum number of possible types (static) of 20, which allows testing both low and high degrees of polymorphism, with patterns featuring as low as 1 actual live type at runtime (monomorphic) and as many as 20 (megamorphic [AH96]). Overall, this maximum degree of polymorphism of 20 is representative of behaviors and data we gathered at various sizes (we actually tested all maximum sizes from 1 to 10, then 20, 30, 50 and 90). Shorter static type sizes, which are the most common in real applications, typically lead to more efficient *if sequences* and binary search trees.

Results are presented in figures 6 and 7, that show two different JVMs on the same Pentium platform, as well as in figures 8 and 9 that show the HotSpot client JVM on two different hardware platforms.

On all these graphs, the same 5 benchmarks are tested, resulting in the 5 curves on each graph:

Virtual20 A plain virtual call, implemented with the `invokevirtual` bytecode, that can cope with any number of possible receiver types[1].

BinaryTreeStaticThisarg20 This is a binary tree dispatch, with 20 leaves that are static calls, the receiver object being passed as an explicit argument.

IfSequenceStaticThisarg20 A sequence of ifs containing 20 static leaf calls.

SwitchStaticThisarg20 A Java switch, translated into a `tableswitch` bytecode with 20 cases, each being a static call.

NoCall20 This benchmark contains no call at all, it shows the base cost of the benchmark mechanism (loop and static method call).

The different control structures are tested against 41 execution patterns of the four kinds presented in section 3.4, constant, cyclic, random and stepped, that compose the x axis. The numbers appearing in the pattern name indicate the active range of type IDs for each pattern. Thus `rnd-01-07` is a pattern made of random type IDs between 1 and 7, `step-01-09` is a type ID pattern with 9 steps, from 1 to 9, and `cst-04` is a pattern with constant type ID 4, and so on.

4.1 Observations

Figure 6 shows performance in milliseconds of execution time for the IBM JIT on a Pentium III. Plain *virtual calls* (`invokevirtual`, shown as continuous black curve) appear to be sensitive to the dynamic execution patterns tested. Virtual calls executing constant patterns and stepped patterns take about 700 ms, compared to 1000 ms for cyclic and random patterns. The NoCall20 micro-benchmark executes in 600 ms. Therefore the overhead of virtual calls varies between 100 and 400 ms, a factor of four due only to differences in type patterns. Other JVMs on the Pentium platform show similar ratios (figures 6 and 7). On an UltraSparc III (figure 9), virtual calls appear less sensitive to execution patterns. The constant pattern is executed slightly more efficiently, but a stepped pattern shows the same performance as a random or cyclic pattern. In contrast, stepped patterns with low variability behave well on all Pentium JVMs (figures 6, 7 and 8), with a cost close to that of the constant pattern. Overall, virtual calls tend to be more expensive than other structures especially when the number of different types is small and when the type pattern is cyclic. These results indicate optimization opportunities for JVM implementors.

The performance of *if sequences* depends on the size of the sequence and the rank of ifs exercised, shorter sequences being faster. Short *if sequences* are the most efficient way to implement dynamic dispatch among the tested control structures across all platforms, all JVMs and all execution patterns. Although the precise cutoff point varies, it is safe to consider that *if sequences* up to 4 are a sure win over current implementations of virtual calls. The actual gain in performance varies but can be as high as 52% (including benchmark overhead) on the duomorphic

[1] For Virtual and NoCall, the "20" in the name is only kept for consistency with other benchmark names.

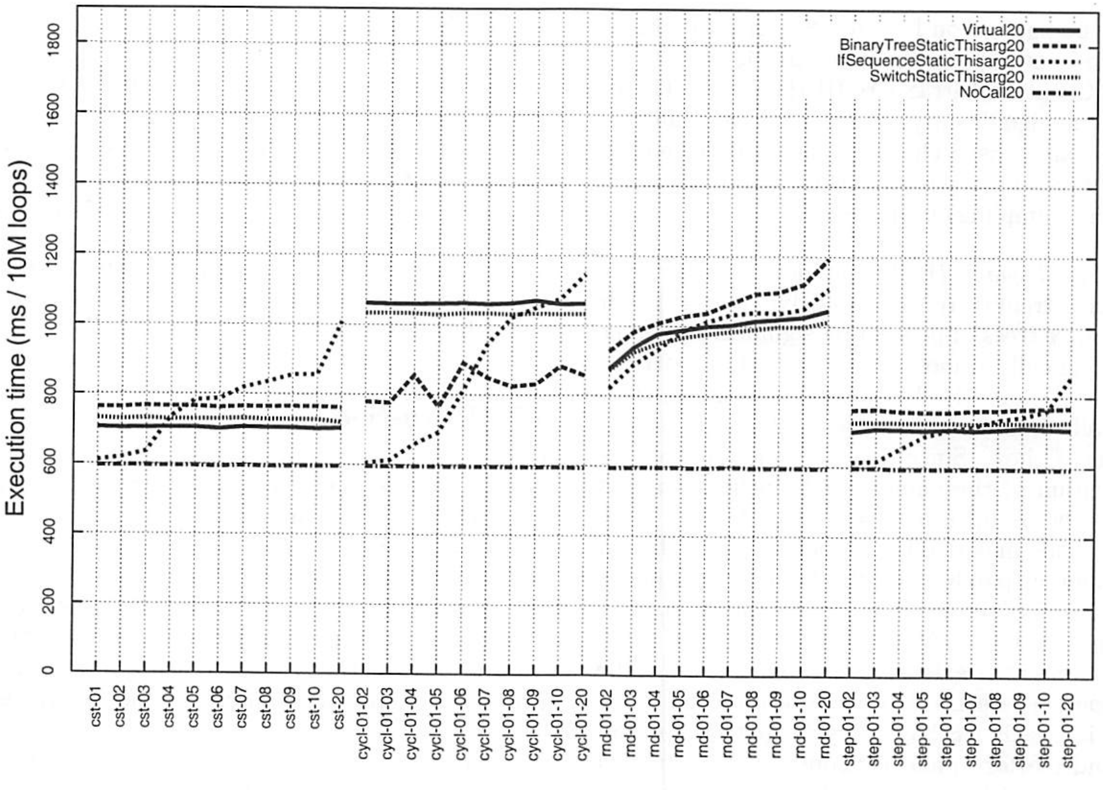


Figure 6: IBM cx130-20010502 on a dual Pentium III

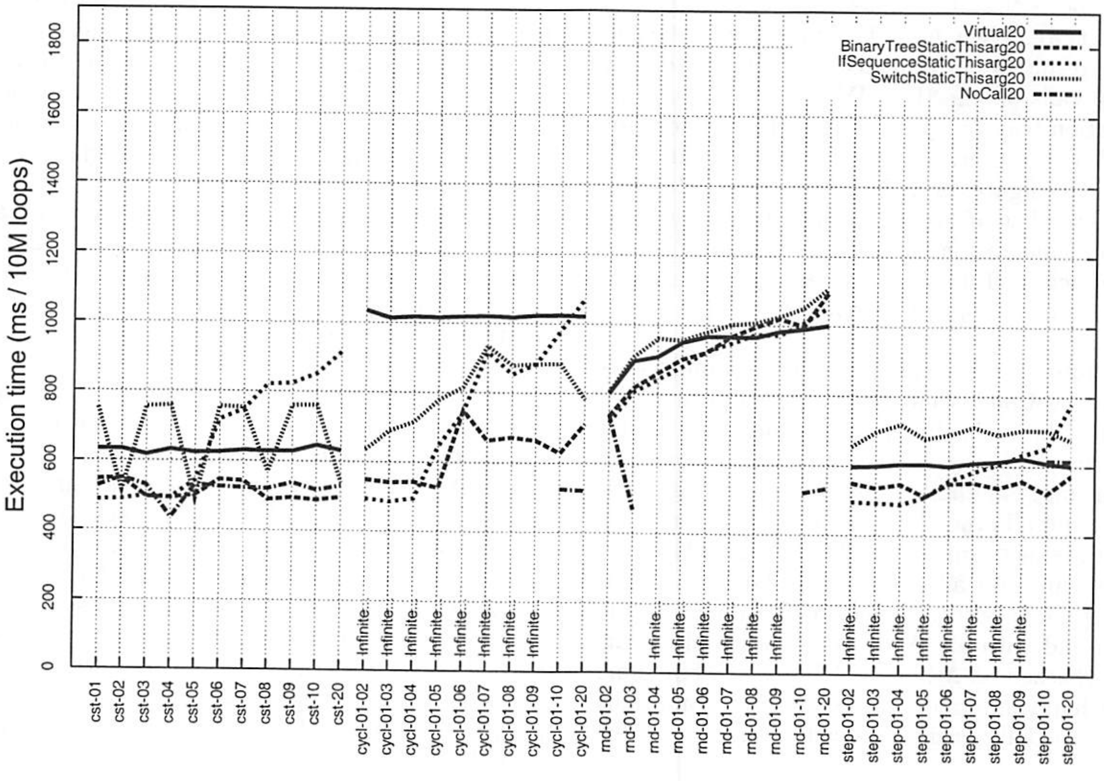


Figure 7: SUN HotSpot Server 1.3.1-b24 on a dual Pentium III

cycl-01-02 pattern on HotSpot Server on Pentium III (figure 7) or 24% on the step-01-02 pattern on HotSpot Client on UltraSparc III (figure 9). Therefore one can significantly optimize the implementation of dynamic dispatch in current JVMs when the number of possible types is known (by static analysis or dynamic sampling) to be small.

Binary tree dispatch (BTD) provides another way to perform strength reduction of dynamic dispatch sites. Binary trees appear to be significantly faster than virtual calls in most cases (all figures, particularly figures 7 and 9). When BTDs are slower than virtual calls, it is generally by a small margin, as figures 6 and 8 show. Since the cost of BTD grows as the logarithm of the number of branches, whereas sequences of *ifs* have a linear cost, BTD is more scalable. This makes BTD a good implementation for dynamic dispatch when the number of types is too large to use simple *if sequences* (above 4 or 8, depending on the JVM and platform), but small enough to prevent extensive code expansion. The cutoff point where BTD become faster than *if sequences* is clearly visible for cyclic patterns on all JVMs and platform, and for constant and stepped patterns in the SUN HotSpot JVMs on both platforms (figures 7, 8 and 9).

Figure 6 shows that Java *dense switches* (bytecodes tableswitch), when used to implement dynamic dispatch, result in performance very similar to that of virtual calls on the IBM JVM, revealing an implementation based on jump tables. In the HotSpot Client JVM however, both on Pentium III and UltraSparc III (figures 8 and 9), tableswitches behave exactly like *if sequences*, which indicates an actual implementation based on sequences of conditional branches. Table switches are therefore unreliable in terms of performance across JVMs.

The "Infinite..." results in figure 7 correspond to executions of NoCall20 that were running forever[2]. We think that this behavior indicates an optimization bug on this particular JVM and platform, since the call of an empty method is an unlikely (but legal) occurrence, and all other JVMs dealt with it correctly. Indeed, the exact same bytecode for NoCall20 is correctly executed on all other JVM-platform combinations, that is with all other JVMs on the same platform and with all JVMs on all other platforms (we also checked on Athlon and Celeron). The same problem happens under the exact same conditions for other NoCall benchmarks with other sizes, but is much less frequent.

Since all our benchmarks are very small and simple and share most of their code, we are confident their Java source code (including the one for NoCall20) is correct. Furthermore, since all the benchmarks are executed correctly on all JVM-platform combinations but one, we trust the javac compiler generated a correct bytecode. We thus suspect some aggressive, non-systematic optimizations by the JVM might be the cause of this issue.

4.2 Discussion

Obviously, using micro-benchmarks focused on dynamic dispatch magnifies the impact of the various dispatch techniques in terms of performance. Although the actual impact on real programs is likely to be smaller, since programs generally do other things than dispatch, our study makes it possible to get a clearer view of what is actually happening. We thus believe that the previous results are an important first step and can already be widely used.

First, these results are important to Java compiler and Java VM designers, when implementing multiple-target control structures such as dynamic dispatch. We show that the performance of dynamic dispatch varies a lot across JVMs, hardware and execution patterns. It is safe to say that dynamic dispatch implementation in current JVMs is not always optimal and can be significantly improved, using mostly known techniques. Direct implementation in the virtual machine is likely to provide the highest payoff.

Second, these results are also useful to Java developers, since they stress differences between the various JVMs, highlighting strengths to take advantage of and weaknesses to avoid, for instance large tableswitches in the HotSpot Client.

Third, our results show that strength reduction of control structures is likely to be beneficial regardless of the hardware and JVM, when the number of possible receiver types can be determined to be small. For numbers of possible types up to 4, *if sequences* are most efficient. Between 4 and 10, binary tree dispatch is generally preferable. For more types, the best implementation is a classical table-based implementation such as currently provided by most JVMs for virtual calls. These are safe, conservative bets, that generally provide a significant improvement and, when not optimal, result only in a small decrease in performance.

Finally, these measurements expose architectural features (especially branch predictors) of the target hard-

[2] "Forever" means for example that such a program was still running after 18 *hours*, instead of a typical execution time below one minute.

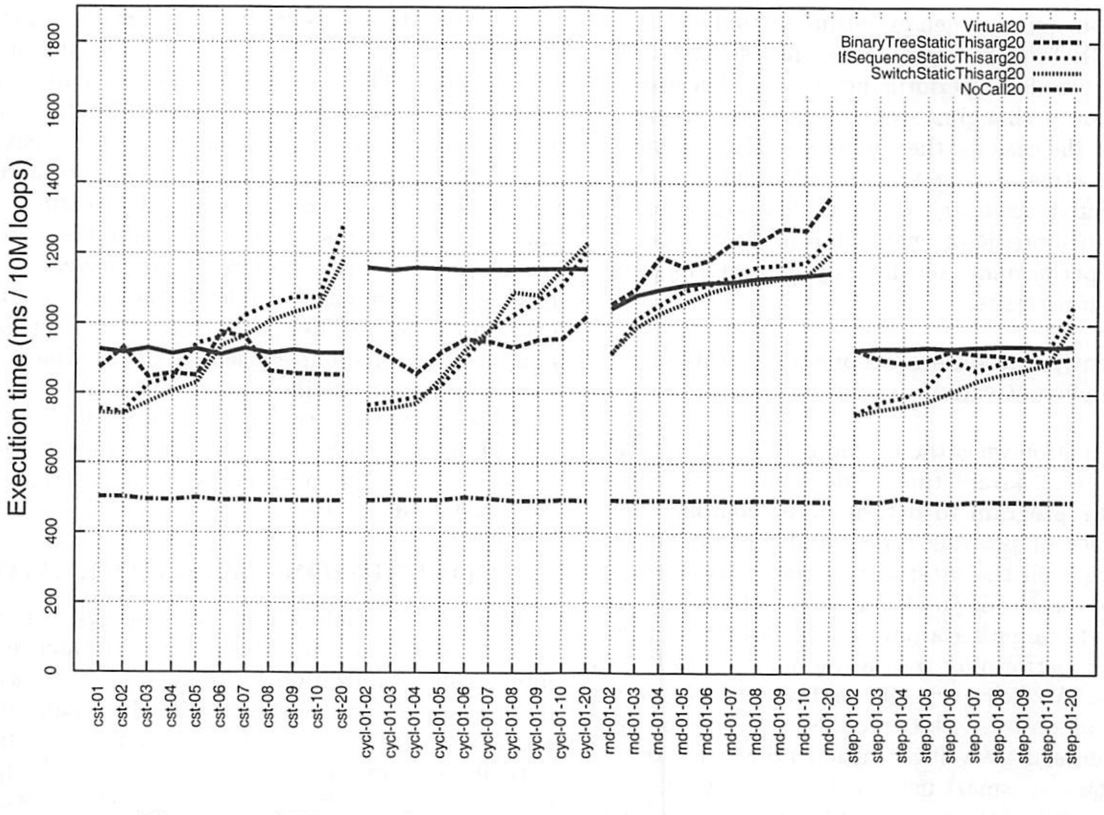


Figure 8: SUN HotSpot Client 1.3.1-b24 on a dual Pentium III

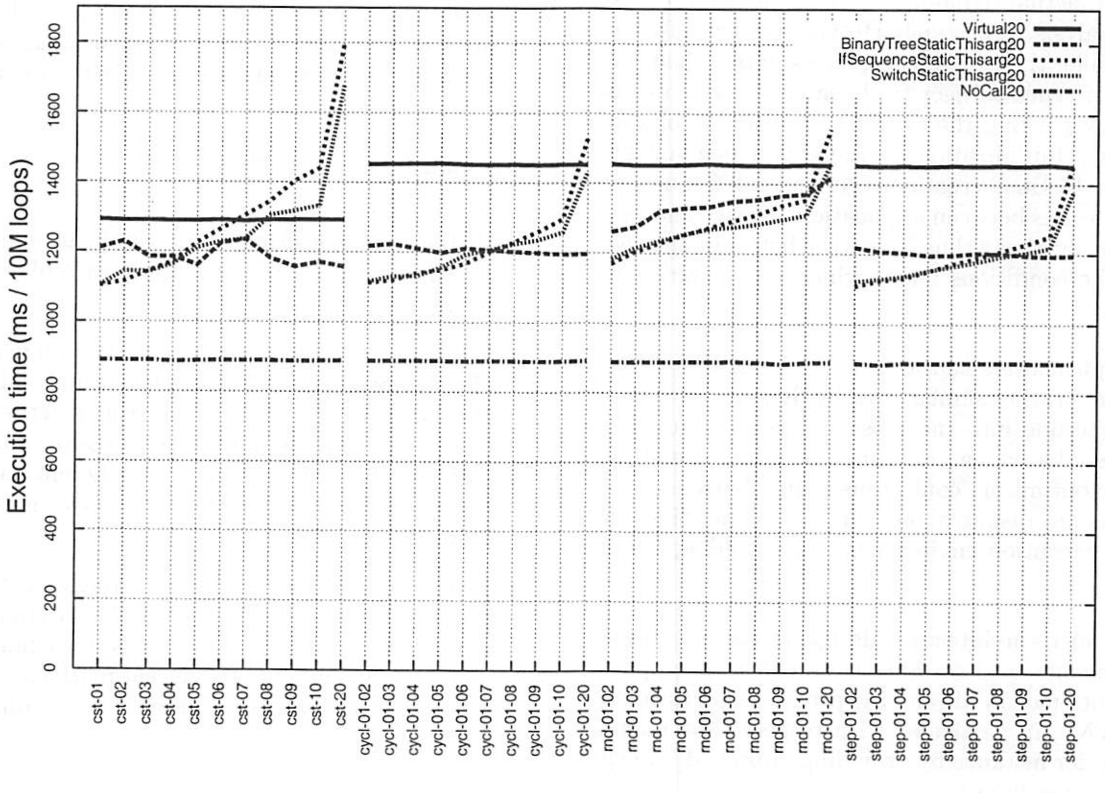


Figure 9: SUN HotSpot Client 1.3.1-b24 on UltraSparc III

ware. For instance, when executing virtual calls the Pentium III branch target buffer ensures that constant patterns have performance nearly identical to that of slowly changing stepped patterns, whereas this is not the case for the UltraSparc III. Similarly, when executing *if sequences*, small cyclic patterns are predicted accurately by the Pentium's conditional branch predictor, which, for all JVMs, results in better performance on small cyclic patterns than on random patterns.

Consequently, the results we provide in this paper can be applied at various levels.

The information we gathered can be used by a static compiler (e.g., javac) that performs a static analysis of the program to determine at compile time the number of possible types, and generate bytecode relying on the most appropriate implementations of dynamic dispatch for each call site, either aggressively targeting a particular platform or conservatively performing transformations for multiple platforms. An extra type ID field might have to be added to all objects, which would lead to per-object space overhead as well as initialization time overhead. However, smart implementations (see [CZ99] for an example in the context of Eiffel) can avoid the need for the type ID field for objects which are not subject to actual dynamic dispatch, as detected by global analysis. The type ID overhead can also be made smaller than an integer, for example when the number of types subject to dispatch fits in 16 or 8 bits, or by packing the type ID in available bits in the objets. Initialization overhead is dependant on the objets lifetime, creation rate, and call frequency, and thus varies between applications. Space and initialization overhead thus have to be better quantified to find the conditions under which each solution is the best.

JVM implementers can also make use of this information in a rather similar way, by dynamically compiling bytecode into the most suitable native code structures, based on program execution statistics. Dynamic optimizers could thus switch between several dynamic dispatch mechanisms, depending on context, execution environment and profiling information.

Finally, micro-architecture designers can use these measurements to determine how to better support the execution of JVMs and the programs that run on those JVMs, in particular with respect to dynamic dispatch, for instance by providing improved branch prediction mechanisms.

As mentioned in section 3.3, all the control structures we studied, except the plain `invokevirtual`, have a size that is proportional to the number of tested types. This number can become quite large, in real OO programs; for example, in the SmallEifel compiler, the maximum arity at a dispatch site is about 50 [ZCC97]. In such cases, an increase in code size could happen, with adverse effects on caches and performance, and thus would have to be mastered. We did not work on this aspect in the present study relying on micro-benchmarks. However, in the SmallEiffel projet, we tackled this issue and used a simple but efficient solution, which consists in factorizing all identical dispatch sites into one or a few dispatching routines ("switch" functions in [ZCC97]). Although we have obtained good results with this technique in Eiffel, we still have to mesure its feasabibility in Java.

5. CONCLUSIONS AND FUTURE WORK

The implementation of dynamic dispatch is important for object-oriented program performance. A number of optimization techniques exist, aimed at de-virtualizing polymorphic calls which can be determined, either at compile-time or runtime, to be actually monomorphic. Complementary techniques, either software- or hardware-based, seek to optimize actual run-time polymorphism as well.

We present a prototype study of various control flow structures for dynamic dispatch in Java, with varying hardware, virtual machine and execution patterns.

Our results clearly show that:

- Virtual call performance is highly dependent on the execution pattern at a particular call site.
- When the call site has a low or medium degree of polymorphism (2-3 target types up to about 10), strength reduction of control structures is likely to improve performance across platforms, using *if sequences* for up to 4 different target types and Binary Tree Dispatch between 4 and 10 different types.
- Processor architecture shines through, more especially on high-performance JVMs: virtual call performance of stepped patterns, for example, is markedly different on different platforms, but does not vary across different JVMs on the same platform.

In future work, we could experiment with more techniques or variants for dynamic dispatch, such as the ones we mentioned in section 3.3, and more platforms (JVM or hardware).

Another area we have to work on is interface dispatch in Java, which is more complex because of multiple interface inheritance, and where some of the techniques we described are not easily applied.

We also plan to more precisely assess the efficiency of the techniques we described by completing our micro-benchmarks suite with larger, real Java programs. This would give more applicable, although less precisely understandable, results.

We also intend to evaluate the impact of these various dispatch techniques with respect to code size and memory footprint, especially for techniques whose code size is proportional to the number of types (*if sequences* and BTD).

We can do so by applying our results either to open-source bytecode optimizers, such as Soot [VRHS+99], or directly to Java Virtual Machines, like the Open VM [Va01], the Jikes Research VM [IBM01] (formerly named Jalapeño) or the SableVM [GH01].

Acknowledgements

We thank Laurie Hendren and Feng Qian, who helped us in early stages of our experiments. We thank everyone who commented on the poster presentation of this work at OOPSLA 2001. We are also grateful to Wade Holst and Raimondas Lencevicius who commented on early versions of this paper, and Matthew Holly who proofread it. Finally, we thank the anonymous reviewers for their valuable comments and suggestions.

This research was supported by NSERC and FCAR (Canada), and INRIA (France).

6. REFERENCES

[AH96] Gerald Aigner and Urs Hölzle. Eliminating Virtual Function Calls in C++ Programs. In *10th European Conference on Object-Oriented Programming (ECOOP'96)*, volume 1098 of *Lecture Notes in Computer Science*, pages 142–166. Springer-Verlag, 1996.

[BS96] David F. Bacon and Peter F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *11th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'96)*, pages 324–341. ACM Press, 1996.

[CC99] Craig Chambers and Weimin Chen. Efficient Multiple and Predicated Dispatching. In *14th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99)*, pages 238–255. ACM Press, October 1999.

[CHP98] Po-Yung Chang, Eric Hao, and Yale N. Patt. Target Prediction for Indirect Jumps. In *1997 International Symposium on Computer Architecture (ISCA'97)*, July 1998.

[CZ99] Dominique Colnet and Olivier Zendra. Optimizations of Eiffel programs: SmallEiffel, The GNU Eiffel Compiler. In *29th conference on Technology of Object-Oriented Languages and Systems (TOOLS Europe'99)*, pages 341–350. IEEE Computer Society, June 1999.

[DDG+96] Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov, and Craig Chambers. Vortex: An Optimizing Compiler for Object-Oriented Languages. In *11th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'96)*, pages 83–100. ACM Press, 1996.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *9th European Conference on Object-Oriented Programming (ECOOP'95)*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101. Springer-Verlag, 1995.

[DH96] Karel Driesen and Urs Hölzle. The Direct Cost of Virtual Function Calls in C++. In *11th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'96)*, pages 306–323. ACM Press, 1996.

[DH98a] Karel Driesen and Urs Hölzle. Accurate Indirect Branch Prediction. In *1998 International Symposium on Computer Architecture (ISCA'98)*, July 1998.

[DH98b] Karel Driesen and Urs Hölzle. The Cascaded Predictor: Economical and Adaptive Branch Target Prediction. In *Micro'98 Conference*, pages 249–258, December 1998.

[DHV95] Karel Driesen, Urs Hölzle, and Jan Vitek. Message Dispatch on Pipelined Processors. In *9th European Conference on Object-Oriented Programming (ECOOP'95)*, volume 952 of *Lecture Notes in Computer Science*, pages 253–282. Springer-Verlag, 1995.

[DLM+00] Karel Driesen, Patrick Lam, Jerome Miecznikowski, Feng Qian, and Derek Rayside. On the Predictability of Invoke Targets in Java Byte Code. In *2nd Annual Workshop on Hardware Support for Objects and Microarchitectures for Java*, pages 6–10, September 2000.

[Dri01] Karel Driesen. *Efficient Polymorphic Calls.* The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 2001.

[DS84] Peter L. Deutsch and Alan Schiffman. Efficient Implementation of the Smalltalk-80 System. In *11th Annual ACM Symposium on the Principles of Programming Languages (POPL'84)*. ACM Press, 1984.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, Massachusetts, 1990.

[GH01] Etienne Gagnon and Laurie Hendren. SableVM: A Research Framework for the Efficient Execution of Java Bytecode. In *1st Java Virtual Machine Research and Technology Symposium (JVM '01)*, pages 27–39. The USENIX Association, April 2001.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification.* The Java Series. Addison-Wesley, 2000. Second Edition.

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches. In *5th European Conference on Object-Oriented Programming (ECOOP'91)*, volume 512 of *Lecture Notes in Computer Science*, pages 21–38. Springer-Verlag, 1991.

[HU94] Urs Hölzle and David Ungar. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. In *1994 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94)*, volume 29 of *SIGPLAN Notices*, pages 326–336. ACM Press, 1994.

[IBM01] IBM Research - Jalapeño Project. The Jikes Research Virtual Machine. `http://www.ibm.com/developerworks/oss/jikesrvm`, 2001.

[Lis88] Barbara Liskov. Data Abstraction and Hierarchy. In *Special issue: Addendum to the proceedings of OOPSLA'87*, volume 23 of *SIGPLAN Notices*, pages 17–34. ACM Press, May 1988.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Language Specification.* The Java Series. Addison-Wesley, 1999. Second Edition.

[SHR+00] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical Virtual Method Call Resolution for Java. In *15th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2000)*, volume 35, pages 264–280. ACM Press, October 2000.

[SLCM99] Ulrik Pagh Schultz, Julia L. Lawall, Charles Consel, and Gilles Muller. Towards Automatic Specialization of Java Programs. In *13th European Conference on Object-Oriented Programming (ECOOP'99)*, volume 1628 of *Lecture Notes in Computer Science*, pages 367–390. Springer-Verlag, 1999.

[UP87] David M. Ungar and David A. Patterson. What Price Smalltalk ? *IEEE Computer Society*, 20(1), January 1987.

[Va01] Jan Vitel and al. The Open Virtual Machine Framework. `http://www.ovmj.org`, 2001.

[VRHS+99] Raja Vallee-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java Optimization Framework. In *CASCON 1999*, pages 125–135, 1999.

[ZCC97] Olivier Zendra, Dominique Colnet, and Suzanne Collin. Efficient Dynamic Dispatch without Virtual Function Tables. The SmallEiffel Compiler. In *12th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97)*, volume 32, pages 125–141. ACM Press, October 1997.

A Just-In-Time compiler for memory constrained low-power devices

Nik Shaylor
Sun Microsystems Laboratories
901 San Antonio Road
Palo Alto, CA 94303
USA

nik.shaylor@sun.com

Abstract

Typical just-in-time compilers for the Java™ platform are often too large and too slow to be used in small computing devices such as cell phones or PDAs. In contrast, the JIT described here was targeted for such devices, and was built on Sun Microsystem's KVM product. Key to making the JIT effective were: a pre-compilation transformation of the bytecodes to make compilation easier; compilation of only a subset of the bytecodes to make the production of the system simpler; quick and simple management of the JIT code buffer; and an effective heap comparison technique that greatly aided debugging of generated code. The JIT speeded up execution by a factor of between 5.7 and 10.7. Its implementation required only 60KB of ARM machine code.

1. Introduction

Slow and space-constrained computing devices have tended not to include virtual computing technology. The advent of Java 2 Platform, Micro Edition Connected Limited Device Configuration (CLDC) [1] has changed this because its small size, and natural portability, has attracted a number of cell phone manufactures to standardize on the Java language as the programming language for third party software.

The Java programs being written for these small systems are often not compute intensive, and so can be satisfactorily executed using a simple interpreter loop written in C or assembly language. Nonetheless, a growing number of these programs are games that require better execution performance than can be provided by interpretation alone. Dynamically compiling bytecodes into native machine code addresses this problem. However, different techniques to those used in larger system are needed to make this feasible.

2. Background

2.1 Traditional JIT compilation strategies

JIT compilation has been used in many programming environments [1][2]. This technique has mostly been used in desktop or server class systems, albeit in different ways. Servers typically benefit from high quality code generation, whereas desktop systems tend to be optimized for reduced program startup latency and better interactive response. To avoid the overhead of compiling and optimizing all an application's classes at runtime, a number of incremental compilation strategies have evolved. Many use an interpreter and a compiler. Others use a number of compilers with different levels of optimization. The general strategy of only compiling the "hot" parts of an application will often result in only a small percentage of an application being compiled, thus saving considerable compilation time.

2.2 Typical J2ME programs

In sharp contrast to larger Java systems the number of classes included in the average J2ME program is far lower, typically less than 50. Compute intensive games are also typically small, which lessens the importance of heuristics that select only the performance critical parts of an application for compilation.

3. Design

The system described here used as its foundation the KVM, a small footprint JVM designed specifically for memory constrained devices.

3.1 Interaction with compiled code

Because the JIT was implemented on top of an existing virtual machine, it was easy to build a compiler that produced native code for only some bytecodes, with the interpreter handling the rest. The compiler was thus straightforwardly targeted to the bytecode subset that produced the greatest increase in performance. There were four reasons why a JIT for the complete bytecode set was not implemented:

> 1, Thread context switching would have had to be performed whilst executing generated native code. This would have added complexity to code generation, runtime support, and the base KVM code. By only performing context switching in the interpreter no changes were made to the way the thread scheduling was done in KVM.
>
> 2, The generated machine code would have needed to be more rigorous in the way it dealt with error conditions and other exceptional conditions. As it is, the machine code only needs to check for error conditions. When they occur the error handling bytecodes can be then executed by the interpreter, which then can deal with the details of how the error should be processed.
>
> 3, A complete JIT would have required more complicated interactions between the generated machine code and the virtual machine as a whole. For example, the generated machine code could cause the compiler, class loader, garbage collector, or native code to run. In retrospect some of these restrictions were not strictly necessary, but the system probably has fewer undiscovered bugs, and it does not seem to have limited the performance of the type of compute-intensive software that is the target of the design.
>
> 4, A debugging technique (discussed below) was used which could not have been employed so easily with a complete JIT.

Therefore the system was designed to allow execution to pass from the compiled code to the interpreter at any time, and also for the interpreter to be able return to generated code in a timely fashion. Additionally, to keep the interpreter from getting trapped in a long loop of bytecodes it was necessary to be able to return to compiled code in the middle of a method as well as at the start.

The basic interpreter loop is as follows:

```
Start:

        Try to enter compiled code.

        Interpret the next bytecode.

        goto Start.
```

The attempt to enter compiled code has several parts. First, if the current method has not been compiled then checks are performed to determine if it can be. Compilation may not be possible for one of the following reasons:

> 1, The method is a *native* method. (A function written in C or assembler).
>
> 2, The method has more than a certain number of parameters or local variables, is unusually large, or has some other condition that basically only occurs in test code and is too troublesome to deal with.
>
> 3, There is no available memory for more compiled code.

If a method can be compiled then a jump table is also produced that contains the machine code addresses of a number of entry points into the compiled code, and their corresponding bytecodes addresses. When attempting to re-enter compiled code a search is then made of this table to see if the address of the current bytecode is present. If it is then the corresponding machine code address is found and the compiled code is entered at this place. The jump table contains the addresses of all backward branch targets so it is therefore not possible to be stuck in a loop in the interpreter.

If compiled code is entered, then some amount of computation will result. The return to the interpreter is simply done by having the compiled code update the interpreter's state (instruction pointer, local variables, etc.), and then exiting back to the interpreter loop. The interpreter will then start executing the method where the compiled code left off.

There are several conditions that will cause control to be returned to the interpreter. Some of these conditions exist because the JIT lets the interpreter deal with complex situations (such as exceptions, synchronization, or garbage collection):

1, A native function was called.

2, The thread was blocked because of a monitor enter operation.

3, An object could not be created without running the garbage collector.

4, A method was called but a monitor or an activation record could not be created without running the garbage collector.

5, An operation was attempted that required a class to be initialized.

6, The start of an exception handler was reached.

7, An exception or error occurred. The interpreter always processes these.

8, The part of a method was reached for which no corresponding machine code could be generated.

9, A function was called for which there was no compiled code.

10, A method return was executed but there was no record of the where in compiled code to return to.

11, A method return was executed but there was no compiled code to return to because the code buffer had been flushed.

3.2 Register allocation

It is important that compiled code can be entered at places other than the start of a method. Without this a long running method might be prevented from entering compiled code for a long time. To support this, a table of entry points is generated that contains, at least, an entry for the start of the method and all the backward branch targets.

There is, however, complexity associated with entering compiled code at an arbitrary branch target, because the correct machine state must be established before this can be done. This is complex because a Java interpreter is, most naturally, a stack based computing machine, whereas the generated machine code for most computer hardware will, most profitably, use a register-oriented model.

This problem is neatly avoided in this system by preprocessing the bytecodes into a form where there is never anything on the interpreter's evaluation stack at a place where control might be transferred between the interpreter and compiled code. This essentially means that there is no evaluation stack, and to compensate, additional, local variables are used instead to hold intermediate expression results. This creates a simple one-to-one correspondence between the registers used in compiled code for expression evaluation and the local variables on the Java stack.

In the ARM implementation of the JIT there are 12 registers used by the compiler to represent local variables. Three of these are regarded as general temporary registers. The other nine are used to shadow the first nine local variables of the activation record for the method being executed. The three general registers are used in the cases where a local variable is required that is not one of the first nine. There are three because of the three-address nature of the ARM instruction set.

This simple form of register allocation worked surprisingly well in the tests made upon the system because they rarely used more than nine local variables. Nevertheless, this register allocation is not, in itself, a complete solution.

3.3 Pre-compilation by bytecode transformation

A key part of the JIT design was to split the compilation process into two passes. The first pass transforms the standard, stack-based bytecodes into a simple 3-address intermediate representation in which all temporary expression results are placed into new local variables instead of entries on an evaluation stack. The second pass converts this three-address form into native machine code.

Because of the relatively small number of methods used by programs in small devices, the system simply converts all methods in all classes as they are loaded. Although some care has been taken to make this process fairly fast, it has not been optimized, as its efficiency does not appear to be an important factor. In testing, there have typically only been between 200-400 methods (including the methods of system classes). What *is* an important factor is the amount of temporary memory needed to do the conversion. Consequently, the system only holds one basic block's worth of IR at a time, discarding it after the block is converted.

The basic process being performed in the first pass will be familiar to the author of the simplest compiler. The instructions that use values from the stack are linked to the instructions defining these values; local variables are then assigned the temporary values that connect the instructions together. A number of standard optimizations are then employed.

The resulting 3-address intermediate representation is then converted back into standard bytecodes. Figure 1 shows the bytecode sequence for the expression $a = 1 - (b * c)$ before transformation. Figure 2 shows the corresponding bytecode sequence after transformation. It can be seen that a new local variable (number 4) has been added to hold the intermediate result. This type of transformation typically causes the code length to increase by about 30%.

```
iconst_1
iload_1
iload_2
imul
isub
istore_3
```

Figure 1. The bytecodes for the expression *a = 1 – (b * c).*

```
iload_1
iload_2
imul
istore_4
iconst_1
iload_4
isub
istore_3
```

Figure 2. The bytecodes after transformation

It is important to note here is that transformation essentially eliminates the evaluation stack, even though the output of transformation is bytecodes. The instruction granularity of the IR is preserved in the bytecodes, so that the evaluation stack is only used *within* the bytecode implementation of a single IR instruction. Switching between the interpreter and native code is only done at the boundary of an IR instruction, so the use of the stack is never a factor because it is always empty at these points.

The transformation process is currently not very sophisticated, but it is clearly feasible to use simple static analysis to order the local variables so that the most used ones would be assigned to machine registers by the compiler.

An attractive benefit of this transformation process is that it is possible to perform it ahead of time. However, one problem that arises when it is done ahead of time is that the transformer needs to know how many local variables can be mapped into registers. Since not all target architectures are the same, the transformed result cannot run optimally on all of them. A solution would be to do the transformation on the target device as a part of application installation, perhaps done as a background activity (only while the devices is having its battery charged for example), or as a one-time operation by the virtual machine with the results being saved for future execution.

3.4 Code Generation

A basic hypothesis of the design of the JIT was that code generation can be done very quickly. This means that a relatively simple strategy can be used for managing the memory buffer for the generated code, since, if code is discarded non-optimally, it can easily be regenerated. Therefore a single fixed-sized buffer is used for the generated machine code. When it becomes full, the entire contents are discarded. This strategy makes simple a number of standard optimizations. For example, in common with the HotSpot virtual machine[4], monomorphic method invocations are generated for calls to methods where there is only one known receiver type at code generation time. Simply discarding the entire code buffer when the relevant method is subsequently subclassed makes this simple strategy completely safe.

As mentioned above, the current implementation does not retain the IR created by the bytecode transformer. The code generator works by parsing the bytecodes back into three-address form, then emitting the corresponding machine code. On a StrongARM PDA running at 206MHz the process of compiling 250 methods takes less than 100 ms. An experiment using a version of the JIT that generates Pentium instructions reveals the compilation cost to be 75 Pentium cycles for each byte of the input bytecode stream. This experiment has not yet been done on the ARM system; one would expect it to be faster because the Pentium code generator contains a more sophisticated register allocation algorithm than the ARM implementation. Nevertheless, even at 75 cycles per byte it is more than an order of magnitude faster then most other JIT systems. Obviously this comparison is not really fair because it does not take into account the time taken by the bytecode transformer. However, by accepting a small pause when starting up a program (typically only 10% extra startup time) the code generation time is made virtually insignificant, and by saving the transformed bytecodes for future invocations the transformation cost would only paid once. Seventy-five cycles is roughly the time needed to interpret two bytecodes on KVM, so the cost of code generation is quickly amortized during execution. When the size of the code buffer is reduced to about 60% of the code working set of a program the compiler has to run very frequently to regenerate code, because the buffer is constantly being filled and then flushed. Nevertheless, even in this case the execution time is typically two to three times faster than running the bytecodes using the interpreter alone.

3.5 The format of activation records

The KVM implementation is a very literal encoding of the Java virtual machine specification. Five significant virtual machine registers are maintained, and the processes of method invocation and method return are quite lengthy. It was found that replicating this very literal behavior in compiled code slowed down method invocation significantly, so a different calling convention is used in compiled code. This calling convention was designed so that return to the interpreter can occur at almost any place in a method. This led to the implementation of a stack frame *converter* that can change the activation records from the form created by compiled code back to the form used by the interpreter. It also necessitated that local variables held in machine registers be saved and loaded into the activation record by the caller instead of by the callee (which would be more efficient because registers unused by the callee do not require saving and restoring). This issue seems not to have hurt performance too badly, probably because all the registers can be saved using a single ARM instruction. The stack frame converter is not particularly complex, but various performance tradeoffs were encountered during its implementation. At one point the normal calling procedure was made faster at the cost of more work being done by the stack frame converter. This considerably lengthened the time it took to switch back to the interpreter. This in turn caused the frequency of this operation to be more of a factor in overall performance. It was no longer possible to 'quickly' switch back to the interpreter to execute an unimplemented bytecode. In early versions of the implementation, execution would switch hundreds of thousands times per second. It thus became necessary to compile many more types of bytecodes to get the performance up. In the end the benchmark programs only switch about every 10 milliseconds.

3.6 Threading issues

Since the KVM is relatively simple and portable, it has its own internal “green” thread implementation. This made possible a considerably simpler JIT design because there are no native thread preemption or SMP issues to be concerned with.

However, thread switching is an issue. In the current JIT implementation, if a compiled thread goes into an infinite loop the virtual machine will hang forever. Although this does not appear to contradict the Java language specification it is not an acceptable situation. A solution to this would be to dedicate a machine register to be used as a counter decremented at each backward branch. When the counter reaches zero execution would pass to the interpreter so that a context switch could be performed.

3.7 Native methods

Native methods are primitive functions that are usually written in C or assembler. In KVM they naturally fall into two categories, those that take a "long" period of time because they are (typically) waiting for I/O to complete, and those that do not. The former are normally coded in such a way as to cause KVM to context switch, if possible, to another thread of execution. For this reason, compiled code cannot, generally, call native methods directly. However, it was found to be very important for compiled code to be able to call certain non-blocking functions for performance reasons. The compiler was modified to generate direct calls to the code for `System.arraycopy()` and to the graphic drawing functions (although the latter were not called when running the benchmark programs).

3.8 Debugging

The implementation was quite easy to debug due to a new dual execution feature, which enabled straightforward comparison of the results of interpretation and compiled execution. Before the execution of each basic block of machine code the heap was saved. A single basic block's worth of machine code was then executed (this was achieved by the code generator inserting special code to return to the interpreter at the end of each basic block). The heap was then saved a second time, and then the original heap was restored. The same basic block's worth of machine code was then executed on the interpreter using the bytecodes that were used to generate the machine code. Comparing the heaps after two such executions then verified (or not) that the interpreter and the compiled code had done the same work. When a word in object memory was found to differ, the address was noted, and then the garbage collector was run in a special mode that just looked for the given address. If the structure containing it was found the structure was dumped out. Having special annotations for stack structures made identifying corrupt local variables very easy. The vast majority of code generation bugs showed themselves this way.

Although the target platform was the Compaq Pocket PC, most of the debugging was done using the GNU gdb debugger on a workstation. The debugger can be built (with considerable difficulty!) so that it executes code using a machine code simulator. The ARM simulator included with gdb proved very reliable and a great deal faster then the tools available for the Pocket PC, which exhibited a painful delay of 15 to 20 seconds when stepping from one assembly instruction to another.

The combination of using the gdb ARM emulator and running the compiled code one basic block at a time often proved too slow to be practical. In these cases another strategy was employed. Internally, functions with names like Asm_MoveRR() were used to encapsulate the generation of code (this example would generate a register-to-register copy). It was very easy to have these routines generate equivalent machine code for a workstation so the program could be tested without the ARM simulator. Debugging the resulting workstation machine code was not easy, but the much increased speed of execution made it worthwhile.

4. Performance evaluation

The performance of the JIT was measured and compared to the performance of the KVM interpreter. All the optional performance features of the KVM were enabled in order to make the comparison as meaningful as possible. The JIT implementation was based on version 1.0.2 of the KVM. A number of enhancements were subsequently made to this code to incorporate all the significant interpreter performance improvements present in KVM version 1.0.4.

In addition to determining performance improvements due to compilation, measurements were made to investigate the effects of code buffer size on performance. These effects are clearly heavily dependent on the size of the application, but the experiment of particular interest was to see how the system performed when the buffer size was reduced to below that needed to contain the whole application.

4.1 Benchmark tests

In order to evaluate the effectiveness of the design three real-world Java programs were used to test the system.

1, A graphical program specifically written to demonstrate the effect of the JIT. This program renders a number of rotating cubes. (In normal operation, with graphics enabled, it was possible to navigate through this virtual world.)

2, The DeltaBlue constraint solver.

3, An MPEG-1 video decoder.

All three of these programs were run in a special mode in which graphical output was disabled in order that only VM execution time would be measured.

4.2 Results

The three benchmark programs were run on the interpreter-only version of the system and on the JIT-enabled version using a number of different code buffer sizes. Table 1 shows the execution times for the benchmarks in seconds (not every test was performed with the smallest buffer sizes). The same information is presented in figures 3, 4 and 5 in graphical form.

Table 1. Benchmark execution times in seconds

	Cubes	DeltaBlue	MPEG
Interpreter	25.8	6.04	47.5
JIT w/128KB	4.3	1.05	4.4
JIT w/64KB	4.3	1.09	4.8
JIT w/48KB	4.3	1.03	5.8
JIT w/32KB	4.3	3.3	12.5
JIT w/24KB	58	5.3	39
JIT w/20KB	89	83	49
JIT w/18KB	510	89	90
JIT w/16KB	567	163	132
JIT w/14KB		188	168
JIT w/12KB		173	206
JIT w/10KB		170	351
JIT w/8KB		318	

The first observation is that with a large JIT buffer of 128KB the performance of the JIT-enabled version was faster by a factor of between 5.7 and 10.7.

The second observation is that the performance of all the benchmarks remained roughly the same until the buffer size was reduced to 32KB, after which execution times rose significantly due to the code generator running periodically. Code generation has two cost components. One is the basic execution time of the code generator. The other is the cost of converting the stack from the format used by compiled code to the format used by the interpreter, which is done when the code generator is invoked when compiled code is running.

Note how performance degrades differently with the three benchmark programs. The performance of the cubes program drops off sharply below 20KB, the DeltaBlue program demonstrates two distinct plateaus, and the MPEG program shows a more linear decline.

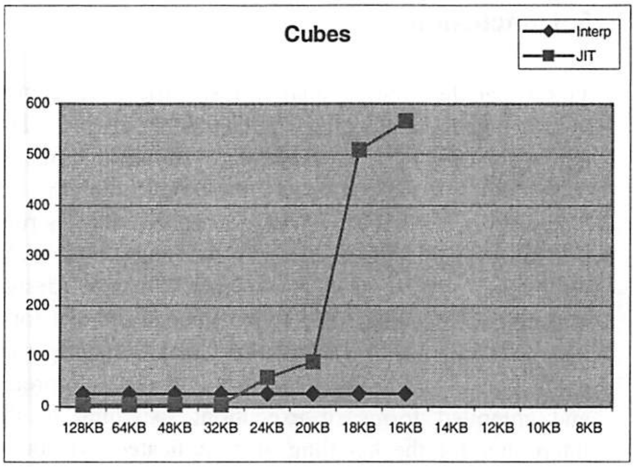


Figure 3. Cubes Execution time.

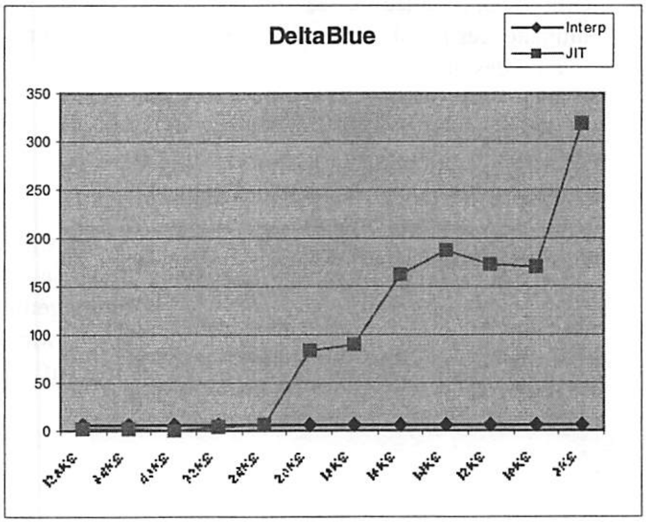


Figure 4. DeltaBlue Execution time.

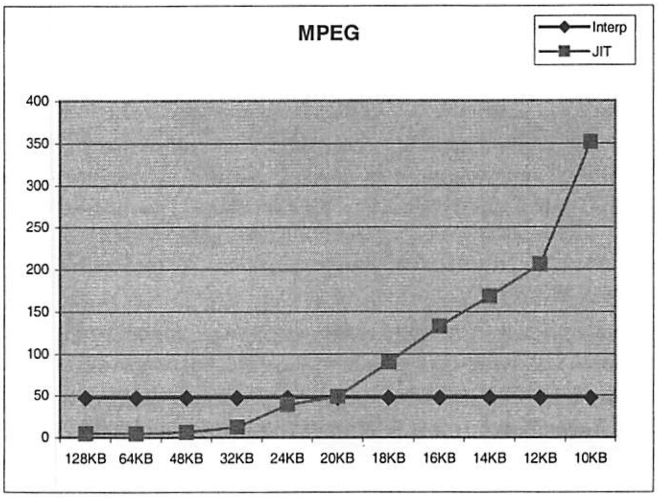


Figure 5. MPEG Execution time.

5. Conclusion

This paper describes a small JIT specifically designed to run Java games on a handheld device. The implementation size is only 60KB of ARM machine code. Although the quality of code produced by the JIT compiler is relatively low, the system nevertheless runs programs five to ten times faster than the KVM interpreter. The noteworthy features of the design include: transforming the input bytecodes into a form that does not use the evaluation stack for temporary results, making it easy to switch between interpreted and compiled forms of execution; switching to the interpreter for the handling of complicated bytecodes, such as those involving exception handling and synchronization; and favoring fast compilation speed over sophisticated code buffer management. The ultimate result of these features has been a simple overall design.

6. Acknowledgements

Thanks go to Kay Neuenhofen who wrote the Cubes programs and the dynamic assembler used in the project, to Kinsley Wong who greatly assisted in getting the ARM port to work, and to Doug Simon, Mario Wolczko, and William Bush in preparation of this paper.

7. References

[1] JSR-000030 J2ME Connected, Limited Device Configuration. http://jcp.org/aboutJava/communityprocess/fnal/jsr030/index.html

[2] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Symposium on principles of Programming Language,* January 1984.

[3] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for Self, a dynamically-typed object-oriented programming language. In *Conference on Programming Language Design and Implementation*, July 1989.

[4] The Java HotSpot Virtual Machine Architecture, March 1998. See whitepaper at http://java.sun.com/products/hotspot/

sEc: A Portable Interpreter Optimizing Technique for Embedded Java Virtual Machine

Venugopal K S
venuks@india.hp.com

Geetha Manjunath

Venkatesh Krishnan

Hewlett-Packard Laboratories, Palo-Alto

Abstract

This paper describes a radical approach to aggressively optimize an embedded Java virtual machine interpretation in a portable way. We call this technique Semantically Enriched Code (***sEc***). The sEc technique can improve the speed of a JVM by orders of magnitude. The sEc technique adapts an embedded Java virtual machine to the demands of a Java application by automatically generating an enhanced virtual machine for every application. The bytecode set of the virtual machine is augmented with new application-specific opcodes, enabling the application to achieve greater performance. Aggressive static or offline optimizations are done to ensure tight coupling between the Java application, Java virtual machine and the underlying hardware. sEc makes an embedded Java virtual machine become *a domain specific Java virtual machine* – a versatility not possible with the hardware.

KEY WORDS: embedded JVM, Java virtual machine, optimization, Interpreter, performance, semantically enriched code, JIT

1 Introduction

An entirely new breed of high-technology embedded products, such as Personal Digital Assistants and E-mail enabled cellular phones, have recently emerged. Manufacturers are deploying embedded Java runtime environments on these devices to enable portability and interoperability with software components. Although embedded processors are inferior in speed and code density, performance expectations are still high. This demand forces all the applications on the embedded environment, including the Java runtime environment, to be *squeezed for performance*. Conventional wisdom holds that the Java virtual machine in an embedded Java runtime environment is a bottleneck. Added to this, the rapid inclusion of new embedded appliances to the market demands rapid availability of *portable* and *efficient* embedded Java environments on these platforms.

A striking characteristic of an embedded appliance is its deployment for a *dedicated* or *mission-specific* purpose. This implies that an application that runs on an embedded environment is relatively static and does not vary as much as on a generic computing environment. This is an opportunity for optimizing the performance of an embedded Java environment for the application.

This paper describes a radical approach to aggressively optimize an embedded Java virtual machine interpretation in a portable way. We call this technique Semantically Enriched Code (***sEc***). The sEc technique can improve the speed of a JVM by orders of magnitude. The sEc technique adapts an embedded Java virtual machine to the demands of a Java application by automatically generating an enhanced virtual machine for every application. The instruction set of the virtual machine is augmented with new application-specific opcodes, enabling the application to achieve greater performance. Aggressive static or offline optimizations are done to ensure tight coupling between the Java application, Java virtual machine, and the underlying hardware. The sEc technique makes an embedded Java virtual machine become *a domain- specific Java virtual machine*.

A goal of the sEc technique is have neither a runtime optimization overhead (as with *just-in-time compilation* (JIT) or *dynamic code generation),* nor to perform exhaustive optimization by ahead of time compilation, thereby losing the dynamic loading capabilities of an application. Our technique provides an intermediary solution. To summarize, the sEc technique makes an embedded Java virtual machine become *a domain specific Java virtual machine* – a versatility that a virtual machine enjoys over hardware processors.

Java[5], JVM[4] and JavaSoft are all trademarks of Sun Microsystems. In the rest of the report, the words JVM, application, Java runtime environment, unless specifically stated otherwise, are assumed to be an embedded JVM, embedded application, or embedded Java runtime environment respectively. Section 2 gives the genesis of the technique and section 3 gives an overview. The full technique is detailed in section 4 followed by implementation and results in section 5. Related work is presented in section 6 followed by conclusion, section 7.

2 The Genesis of sEc:

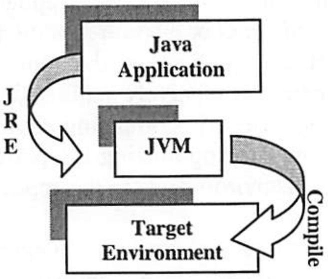


Figure 1: Java Embedded App

The sEc technique is motivated by the salient characteristics exhibited by the three building blocks of a Java application environment: the Java application, the JVM , the target environment, and the two couplings that exist between them as show in the figure-1. We now enumerate the characteristics of these three building blocks.

Java Applications: First, due to their object-oriented nature, applications are method-call intensive. Second, garbage collection frees the program from memory related problems like pointer chasing and dangling pointers. Third, the dynamic behavior of the applications can be characterized by the 80-20 rule of thumb, i.e. 80 percent of the execution time is spent in 20 percent of the application.

Embedded Java Virtual Machine: First, the JVM is a stack-based virtual machine (VM), where the stack is emulated using a heap. The JVM operations are dominated by stack operations. Second, the JVM byte codes have high semantic content compared to the target machine instructions. Third, the JVM is executed on a real machine and the byte code of the application is interpreted and spends 80% of its execution time in the *interpreter* loop. Fourth, the JVM provides the semantics of *dynamic loading* of classes as language feature. This is manifested in the *late binding* model for runtime entities.

Embedded target environment: First, recent embedded processors are register based RISC or CISC machines. Second, an embedded environment is created for a dedicated purpose, which implies applications are relatively static. Third, to keep pace with a fast product cycle environment, it must be both portable or retargetable and efficient.

At the application execution time the above-mentioned traits give rise to two couplings: the *Java runtime environment* (JRE) and *compilation* couplings, as shown by arrows in figure 1. These couplings have the following positive and negative characteristics.

2.1 ***JRE coupling***: This coupling exists between a Java application and the JVM at interpretation time and hence is dynamic in nature. The previously mentioned characteristics of a Java application are manifested by *Pointer-intensive and Call- intensive* nature in this coupling. Indirection of pointers has become the fundamental unifying model of all Object-Oriented runtime environments to support polymorphism and runtime type-check systems [13][14][1], e.g. the dispatch table mechanism used in the implementation of the "invokevirtual" opcode. JVM sub-modules, namely garbage collection and object management also make this coupling pointer- intensive. On the other hand the high semantic content of JVM bytecode causes the implementation of their JVM interpreter action to introduces, on average, one or more function calls per opcode. This makes the JRE coupling exhibit *call-intensive* characteristics.

2.2 ***The Compilation coupling***: This coupling is between the JVM and the target environment and is created during compilation of the JVM source and therefore it is static. Treating a JVM as just another program by the compiler has several disadvantages. First, the JVM features mentioned above introduce imprecise information to the compiler and thereby greatly hamper the optimizations by the compiler. The pointer-intensive characteristics of the JVM source induce data dependencies, leading to imprecision in the compiler analysis required to perform optimization transformations on the JVM code. Second, the pointer-intensive and call-intensive nature of the JVM also introduces control dependencies and hampers inter-procedural analysis. Modern target machines are more efficient when jumping to a constant address than when indirectly fetching an address from a table, such as a dispatch-table, which stalls the instruction-issue and execution pipeline for several cycles[8]. Third, the compiler used to (cross) compile a JVM source is ignorant of the JVM high level semantic constructs and architecture, and therefore fails to exploit precise information (e.g. the semantics of the stack operations) to improve the efficacy of the optimization. Fourth, the late binding model of the JVM also hampers optimizations by deferring the binding to the runtime. For example, the precise information about the binding address of symbols and the branch targets will increase the efficacy of optimizations of the JVM interpreter action code.

The driving philosophies of the sEc technique are (1) to make the application drive the semantic content of the

JVM by creating new opcodes (sEc-opcodes), resulting in an adaptive opcode set for JVM -- creating a domain specific Java virtual machine, (2) to do offline aggressive optimization for speed of the frequent case or frequently executed traces, while exploiting information on runtime constants and (3) to make implementations of the stack-based JVM tightly coupled to the real target (register-based) machines.

3 The sEc solution – an overview

This section gives an overview of the sEc technique, with sections giving details. Functionally, the sEc technique has three major phases namely, (a) sEc detection, (b) sEc Code Generation and (c) sEc embedding, as seen in figure 2.

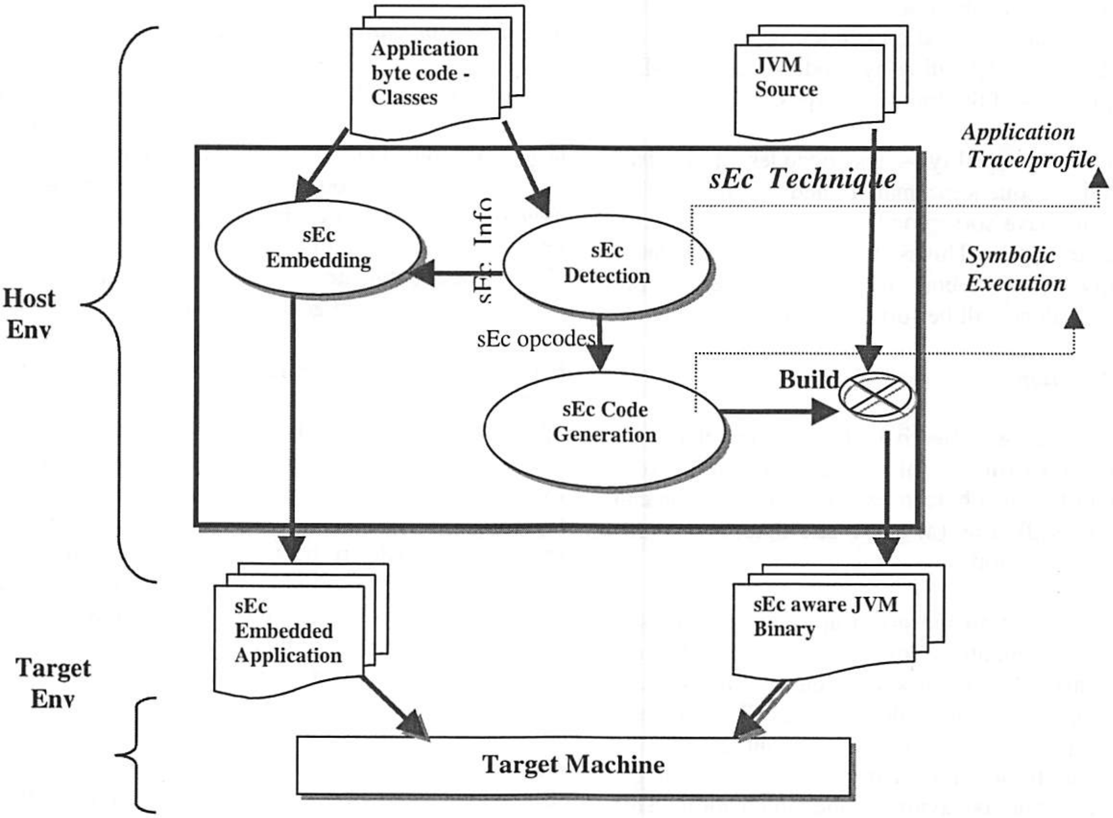


Figure 2: Three phases of the sEc Technique.

(a) *sEc detection phase*: The core of the JVM is the interpreter module, which interprets the bytecodes of the JVM. Interpretation is based on the fundamental pattern of *fetch-decode-execute* and *loop back*. This phase deduces the sEc-opcodes - new JVM bytecodes - from the execution semantics of the Java application. The new sEc-opcodes are synthesized from the stream of an application bytecodes using the application trace / profile or the probabilistic expectations of the execution. In our experiments, we use application profile information.

(b) *sEc code generation*: This phase takes the sEc-opcodes as input and generates efficient 'C' native code, which is smaller and faster than the equivalent JVM 'C' action code for the bytcode sequence represented by the sEc-opcode. This phase not only eliminates the per-opcode overhead of interpreting the sEc-opcodes - fetching and decoding each bytecode - but it also optimizes based on the JVM semantic content of the sEc-opcode. A novel technique called *sEc Symbolic Execution* is devised for sEc-opcode code generation. The generated code will be the JVM interpreter action for the sEc-opcode.

(c) *The sEc Embedding* makes the generic JVM aware of the synthesized sEc-opcodes and embeds the sEc-opcode appropriately in the place of the (Java) bytecode sequence in the application. This can be done either dynamically or statically, at runtime or offline, respectively. Finally, the sEc embedded application is executed using the *sEc-aware JVM* on the target machine for faster execution of the application.

4 The sEc Technique

This section details the design alternatives and phases of the sEc technique. An *sEc-opcode* is defined as a flow-sensitive maximal sequence of computation of Java bytecodes with a candidate based on execution speed using the properties of a dynamic execution. In this definition the basic block (BB), extended basic block and fragment (similar to the BB but with backward branches allowed) in the application trace are candidates for an sEc-opcode. Further more, the definition of equality of sEc-opcodes is based on the equality of the corresponding bytecode sequence. sEc-opcode equality is of the following types:

A. Exact match equality – The opcodes of the two bytecode sequences match and corresponding bytecodes have corresponding matching attributes.
B. Template match - This is similar to the exact match equality above but matching corresponding opcodes alone will be sufficient for a match.

4.1 *sEc Detection*

This phase deduces effective sEc-opcodes that will speedup the Java runtime of a given application from the stream of the Java bytecodes. The sEc detection can further be classified as (a) static sEc detection or (b) dynamic sEc detection.

In static sEc detection the given application is parsed for the most repetitive longest sequence of Java bytecodes, and sEc-opcodes are selected from these based on cost and control flow criteria. This method fails to capture the dynamic semantics of the application, as these patterns may not account for the dominant dynamic behavior of the application. This alternative is better suited for the bytecode compression of the application than for sEc-opcodes synthesis.

Dynamic sEc detection uses the profile of an application generated using representative input as the *best approximation* of the dynamic semantics of the application. Finding the optimal sequences of bytecode to select as an sEc-opcode with respect to speed and space criteria is combinatorially difficult [12]. Hence, the following heuristics based on greedy and non-greedy approaches are used.

A. Greedily select the bytecode sequence which captures the longest repetitive computational sequence of the dynamic bytecodes stream. The disadvantage of this method is that it is insensitive to control flow into the sequence. The overhead of guaranteeing correctness in sequences with multiple branch-targets in sEc-opcode, and Java's precise exceptions cut into the anticipated gain.

B. Non-greedy bytecode sequence selection will deduce the sEc-opcode bytecode sequence under structural constraints like control and data flow. The structural constraints could be the basic block, extended basic block or fragment. These constraints improve the efficacy of sEc-opcode optimization compared to multiple branch targets in the sEc-opcode.

4.2 *sEc optimization and code generation*:

This phase maps the bytecode sequence in the sEc-opcode onto optimized portable native 'C' code for the target machine. This code is the JVM interpreter action code for the sEc-opcode. A unique technique called *sEc Symbolic execution* – an integrated optimizer and code generator - optimizes the sEc-opcode with respect to the JVM semantic domain as well as the target architecture semantic domain, and generates the JVM action code in 'C'. This optimization is effective not only because bytecode dispatch overhead is eliminated for the bytecode in the sEc-opcode, but also because stack operands are folded, redundant local variable accesses are eliminated, and more precise information is available for the offline 'C' compiler optimizer. The resulting 'C' code undergoes further optimization – optimization with respect to the target architecture semantics - by a global optimizing compiler like GCC[10]. This yields efficient sEc-opcode execution resulting in higher coupling of a JVM to the underlying target machine semantics in the resulting sEc-aware JVM.

4.2.1 *sEc-opcode optimization*:

Since the stack based JVM opcodes are emulated over the register-based target machine instructions, the sEc technique allows the optimization of sEc-opcode as listed below.

1. ***Virtual Machine architecture dependent optimizations*:** These optimization techniques are dependent on the virtual machine architecture and its emulation aspects. In summary these optimizations provide efficient storage and access for the sEc-opcode operands and local variables.

- ❑ *Java stack access operands are subsumed:* Within the context of an sEc-opcode, stack operand access is intrinsically subsumed by the efficient 'C' code, eliminating store and load operations on the Java stack frame.
- ❑ *Elimination of redundant Java Local Variable Accesses*: Accesses to the redundant local variable on the Java frame are eliminated by reusing the value in the generated 'C' code. This optimization keeps track of *read-after-read* data dependencies with respect to the JVM architecture across the Java bytecodes that are within the sEc-opcode, and propagates the value.
- ❑ *Elimination of JVM operand stack manipulation operations*: Our studies have shown that using dynamic instruction distributions, 40% of the instructions are related to moving the data between the Java operand stack and local variables, duplicating values on the stack, and constants. For example, consider the bytecodes pop, pop2, dup, dup2, dup_x1, dup2_x1, dup_x1, dup_x2, and swap. This technique keeps track of the stack state in the sEc-opcode and propagates the value to the target operation, thus eliminating the respective bytecode action or the need to generate 'C' code to perform the operation.
- ❑ *Java bytecode is semantically rich*: Within an sEc-opcode, the Java bytecode semantics can be treated as composed of fine-grained sub-operations or predicates. These can be the class, method and field attribute checks. For example, isNativeMethod (method) is a predicate in the bytcode "invokestatic". Runtime checks like exception checks (null value check, array boundary check etc. are also handled). This optimization eliminates these redundant sub-operations within the sEc-opcode. We note here that a common sub-expression in the sEc-opcode is not necessarily possible so in the underlying compiler for the target machine (i.e. pointer aliasing problems arise because of emulating the JVM stack using the heap memory)

2. ***Virtual Machine architecture independent optimizations***: These optimizations are independent of the VM architecture and its implementation, but are limited to the semantic domain of the virtual machine architecture. Some examples of these are common sub-expression elimination within the bytecode sequence of the sEc-opcode, deducing *polymorphic* call points as *monomorphic* call points (method-pointers are compile time constants) and so on.

3. ***Virtual Machine Runtime Bindings***: The *late binding* or *dynamic loading* feature of the Java VM gives rise to new kinds of optimization opportunities based on runtime constants after the late binding process within an sEc-opcode. We term this novel optimization technique ***sEc-rewriting.*** sEc-rewriting is the dynamic self-redirection of the sEc-opcode to an efficient runtime implementation based on runtime constants and bindings. In case of the JVM, the real machine address bound to symbolic information, (e.g. field or branch offset), is constant after it is resolved, or bound, at runtime. Similarly, some predicate or attribute checks outcomes are constant once resolved. The sEc-opcode is aggressively optimized offline for this specialized runtime variant. During sEc-opcode interpretation, the call-point in the method code is rewritten to jump to the specialized implementation in the very first interpretation, taking runtime values as parameters. Every sEc-opcode potentially has a specialized sEc-rewriting opcode variant, and the virtual machine developer has the option of fine-grained control to select them.

4. ***Target architecture dependent and independent optimizations***: These optimizations are in the purview of the underlying real machine semantics or architecture[7]. In the case of RISC architectures, optimizations dependent on the register architecture, like instruction selection, instruction scheduling, register allocation and register assignment are target architecture dependent optimizations. On the other hand, optimizations independent of the register based architecture but constrained by the semantics of the underlying register architecture are termed target architecture independent optimizations. Common sub-expression elimination, copy propagation, and loop invariant code motion, are but a few from the cornucopia of possible optimizations[7]. The sEc technique depends on a global optimizing compiler like GCC to do the optimizations under this category, but provides more precise information to aid the global optimizations.

4.2.2 *sEc Code Generation*

The sEc code generator's aim is to generate efficient retargetable 'C' code for the sEc-opcodes. Clearly, a naive sEc code generator could just concatenate the corresponding JVM interpreter action code of all the individual bytecodes in the given sEc-opcode. This could be visualized *as interpreter switch unrolling* for the respective sEc-opcode, which only eliminates the per-bytecode dispatch overhead. However our code generation technique - sEc Symbolic execution - exploits the optimization techniques explained above. The sEc code generation has to abide by structural constraints that the Java bytecode guarantees, namely, (a) *Stack Invariance of Java Bytecode*: each bytecode must only be executed with the appropriate type and number of arguments on the operand stack or in local variables, regardless of the execution path that leads to its invocation, and (b) *Variant data type of Java stack frame*: At different execution points of the same method, the local variable slot in a Java invocation frame can hold different data types.

1. 'C' local variable allocation of the Java operand and local variable[s] under the sEc code generation constraints enumerated earlier.
2. tracking the state of the JVM to eliminate or subsume sub-actions of the bytecode.
3. ordering the sub-operations like type check, null check etc.
4. ensuring the state of JVM is consistent when control flows into and out of sEc-opcodes.

sE-opcode sequence	*Symbol created*	*sEc Symbolic state* →	*sEc Local Var Table*					*Code Generated in 'C'*
			1	2	3	4	5	
Prologue		//well formed sEc-opcode, no code generated						// No code generated
ALOAD_1	O1	O1	O1					O1 = JLV(1)
ILOAD_3	O2	O1,O2			O2			O2 = JLV(3)
IALOAD	O3	O3						O3 = Macr(O1, O2)
ALOAD_2	O4	O3,O4		O4				O4 = JLV(2)
ILOAD_3	O5	O3,O4,O5			O5			O5 = O2
IALOAD	O6	O3,O6						O6 = Macr(O4, O5)
IMUL	O7	O7						O7 = O3 * O6
ILOAD 5	O8	O7,O8					O8	O8 = f(JLV(1))
IADD	O9	O9						O9 = f(O7,O8))
ISTORE 5							$O9^D$	// No code generated
IINC 3 1					$O5^D$			$O5^D$ = f(O5,1)
Epilogue		//No stack update //consolidated JVM pc update						JLV(3) = O5 JLV(5) = O9 PC = PC + 15

JVM_word *Ptr_lvp = Jvm_stack_base + lvp // Pointer to current Java invocation Frame //	
#define JLV(x) *(Ptr_lvp + x)	***// Reference to Java Local Variable on a Java Frame***
Macr(O1, O2)	***// Macro to fetch $O2^{th}$ element for O1 array object // O1 and O2 are macro parameter.***

Figure 3: Illustration of sEc Symbolic Execution on the sEc-opcode

4.2.3 *sEc Symbolic Execution*

This section defines some terms and explains the code generation constraints and sEc symbolic execution in detail. During the symbolic execution, the bytecodes logically making up the sEc-opcode sequence are symbolically *interpreted* for the purpose of integrated code generation across the sEc-opcode and optimization within the sEc-opcode. This process has knowledge of the JVM stack as well as the target machine architecture, and therefore yields better optimization results. This results in faster execution of the sEc-opcode when interpreted by the sEc-aware Java virtual machine. The following are the essential issues handled in the process:

4.2.4 *An example of the sEc Symbolic Execution*

Consider the bytecode of the source statement, ***sum = a[i] * b[i] + sum,*** represented as a sEc-opcode. The corresponding sequence of bytecodes is *[ALOAD_1, ILOAD_3,IALOAD, ALOAD_2, ILOAD_3, IALOAD, IMUL, ILOAD #5 IADD, ISTORE #5, IINC 3 1]*. The snap shot of the trace of the sEc symbolic execution is shown in figure 3. In this particular sEc-opcode, there is no 'C' code generated for epilogue because the sEc opcode has all the stack operands of the bytecode within the sEc-opcode (well-formed sEc-opcode). We note the following points about the example:

1. The suffix of O's is maintained by the sEc code generator. The string-symbol is generated and pushed on to the symbol stack (Column 3).

2. Although there is a scope to replace O3 by O1 for the "iaload" bytecode, the translator generates the new symbol O3, since replacing O3 by O1 will not add to the ability of the code generator to perform optimizations. However, the GCC compiler easily optimizes the code by eliminating O3.

3. The JLV(n) is a macro which accesses the n^{th} Java local variable from the current Java frame.

4. For certain bytecodes in the sEc-opcode, a new symbol is created with an appropriate suffix and pushed on to the symbolic stack, as shown in figure-3.

5. Every local variable load and store is tracked for redundant usage by using the symbolic local variable table, as shown in the figure. A write to a local variable is tracked using the dirty status in the local variable status.

6. Redundant access to the Java local variable 3 is subsumed by reusing the symbol O2

7. The bytecode "Istore 5" does not cause any code to be generated because it is subsumed by the symbolic transformation in the symbolic local variable table.

8. At end of the sEc-opcode – the epilogue – the JVM runtime locals that are dirty will be written back.

4.3 *The sEc Embedding*

The sEc embedding is a process that embeds the sEc-opcode into the Java application and modifies the generic JVM with the new sEc-opcode interpreter action. This phase is divided into 2 parts (a.) modifying the JVM and (b.) embedding the sEc-opcode into the Java application. The sEc embedding can be performed offline or online.

4.3.1 The JVM modification:

The JVM interpreter loop is modified to detect and execute the new sEc-opcode. In the offline model, modifications related to the JVM source (mainly the interpreter) is done in the host environment of the embedded target and cross built to get the sEc aware JVM. In the online model, a generic JVM is modified to have a stub, whose function is to automatically load the new sEc-opcode when the main interpreter loop traps for the new sEc-opcode. The disadvantage of the later method is the need to dynamically load of module in the runtime environment.

4.3.2 ***Embedding the sEc-opcode into the Java application:***

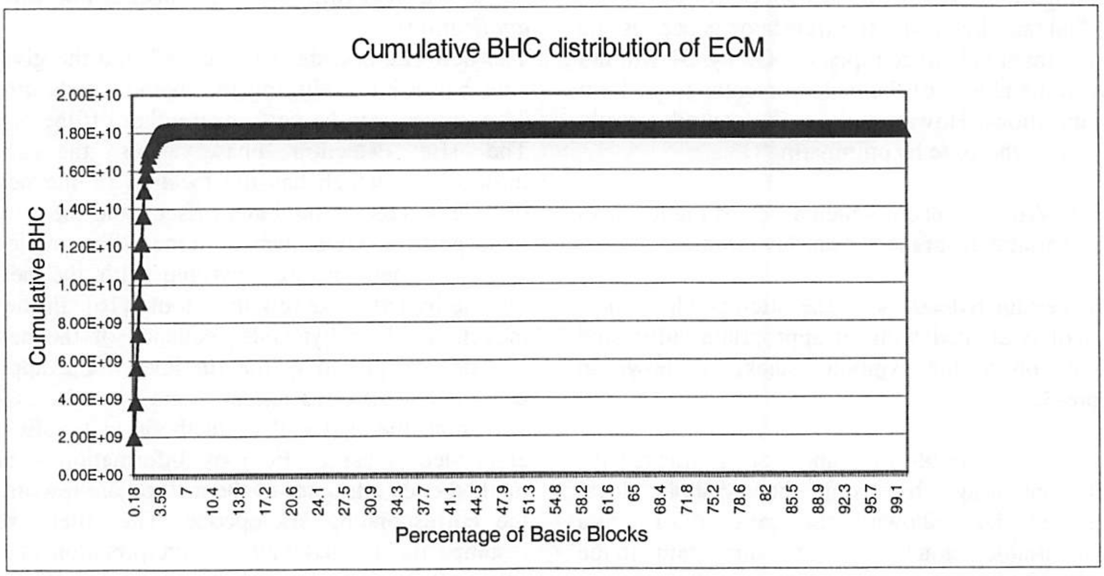
The new sEc-opcode is embedded into the given Java application by modifying the method *code attributes*. This process can be performed either offline or online. The sEc detection phase gives the *sEc-hook* information, which has the location of the new sEc-opcode – class, method and offset - and the size of the replaced bytecode sequence. In the offline model, all of the class methods are rewritten with the new sEc-opcode by bytecode rewriting tools [16]. In the online model, the Java bytecode sequence of the new sEc-opcode is replaced at the runtime of the application using a *special class loader*. The special class loader will track the first call of methods to be sEc-opcode embedded – using sEc-hook information – and the method code attributes of the method are rewritten with the corresponding sEc-opcode. Thereafter execution resumes the normal path of interpretation. The sEc-hook will help the special class loader to pinpoint the method, location and size of the bytecode stream to be rewritten with the corresponding sEc-opcode.

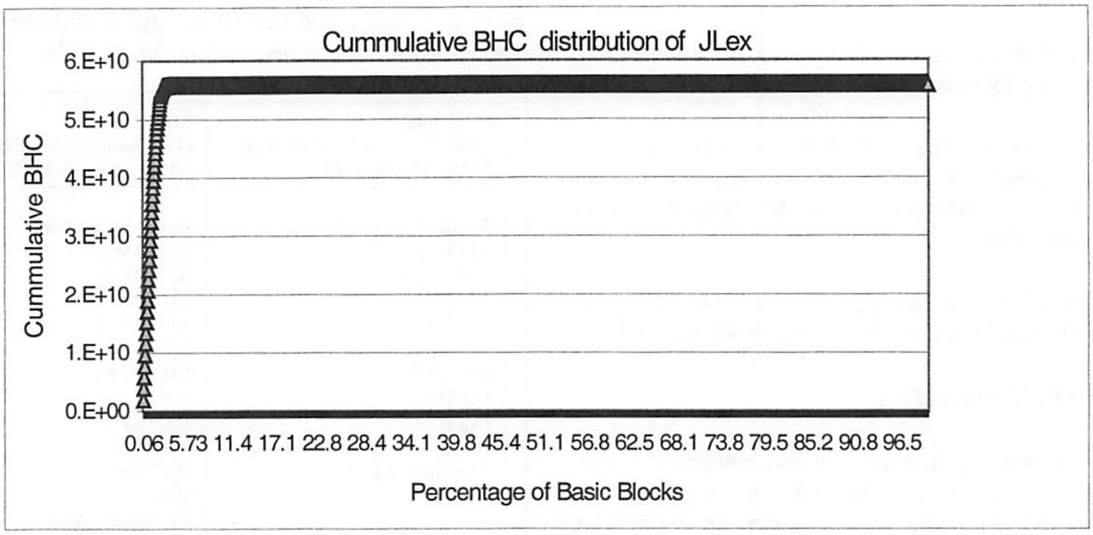
SEC_hook **[Dot, loopit, ()V < offset:31, NUM_INS:9, SIZE:12>**	**SEC_hook** **[Dot, loopit, ()V <offset:54, NUM_INS:11, SIZE:15>**
Begin Basic Block ALOAD_1 ILOAD_3 ALOAD_2 ILOAD_3 BIPUSH DUP_X2 IASTORE IASTORE IINC ***End Basic Block***	***Begin Basic Block*** ALOAD_1 ILOAD_3 IALOAD ALOAD_2 ILOAD_3 IALOAD IMUL ILOAD IADD ISTORE IINC ***End Basic Block***
[1800000]: *Bytecode Hit Count* sEc opcode: sEcopcode_235	***[2200000]: Bytecode Hit Coun*** **sEc opcode: sEcopcode_236**

Figure 4: Bytecode sequence of the sEc-opcodes.

5 Implementation and results

This section gives some preliminary results of the sEc technique. An in-house research JVM [1] was used for the sEc technique and the instrumentation. The host environment was HP-UX 10.20 and the embedded target environment is an NS486 based custom embedded board. The GCC compiler was used for cross compilation.





Dynamic sEc detection with non-greedy heuristics and the BB structural constraint - bytecode patterns are limited to basic blocks - was applied to the benchmarks below. Further selection of the bytecode sequence for the sEc-opcode was based on the *BHC* metric (Bytecode Hit Count) a product of the execution frequency of the BB and the number of the bytecode in the BB. The JVM is instrumented to obtain the BHC and sEc-hook information for every BB.

Fine grained space and speed tradeoff of the JVM:
The Benchmarks ECM (Embedded Caffine Mark) and Jlex (Java lexical analyzer) were used to study the effect of the sEc-opcode on the dynamic semantics of applications and their impact on the speed and space of the applications. Measurements of these are given below. The cumulative BHC chart of ECM shows 6% of BBs accounts for 99.97% of the total BHC for ECM. This amounts to 34 BBs and the probable sEc-opcodes for the ECM application. Similarly, 2.5% of BBs cover 98% of the BHC in Jlex. This amounts to 40 BBs and these are the most probable candidates for sEc-opcodes. A similar graph of JVM code size for every addition of a sEcopcode can be done. This gives the embedded JVM developer the flexibility to do a fine-grained quantitative tradeoff between the application speed and the target space constraints. We note that the number of extended BBs and the fragments that account for the most execution time will be less than the number of BBs using the basic-block structural criteria.

JVM Speedup – Some performance results:
The dot product benchmark was used for the study of the speedup of the JVM. The application was subjected to sEc-detection as detailed earlier. We only considered the two basic block bytecode sequences shown in figure

4 as sEc-opcodes, namely sEc-opcode_235 and sEc-opcode_236 – these sequences had the highest BHC. The sEc-opcode_235 and sEc-opcode_236 are subjected to the sEc code generation algorithm - sEc symbolic execution – performed manually with the JVM dependent optimizations. The resulting 'C' code was used to augment the generic JVM.

The JVM was modified offline to be aware of the new sEc-opcodes. The action 'C' code, was embedded into the JVM interpreter loop and JVM specific changes were made to recognize the new sEc-opcodes. Then, the JVM was built using the GCC compiler with highest level of optimization enabled.

Online sEc-opcode embedding was adopted to replace the sequence of bytecode comprising the sEc-opcode with the sEc-opcode proper. A new class loader was introduced into the JVM to read the sEc-hook information and track all of the loaded classes for embedding the sEc-opcodes at the sEc-hook specified locations. This modification resulted in a speedup of *300%*. We note that only 2 sEc-opcodes were considered for experimentation purposes. Also the bytecode sequences in the sEc-opcodes are less rich in semantics compared to semantically rich opcodes like those for object management and method invocation etc (which would enable more scope to optimize).

6 Related Work

Optimizing a Java application for speed has become an active research area. Broadly, this research can be classified as follows:

Interpreter Techniques: Bringing traditional compiler techniques to the runtime environment will not be a viable solution for resource constrained embedded Java deployment. Studies have been done to improve interpreter techniques as in [19][20]. All of these approaches exploit a variation of threading to improve the per-opcode overhead by removing the opcode dispatch overhead. The sEc technique is a portable, static-optimization interpreter technique. To the best of our knowledge, the sEc technique is the first JVM interpreter optimizing technique of its kind. Compared to the above interpreter techniques, the sEc technique not only removes the per-opcode overhead in the sEc-opcode but also does aggressive optimization in the sEc code generation phase. Unique to the sEc technique, it divides the sEc-opcode optimizations between Java virtual machine dependent and independent optimizations, and JVM runtime binding optimizations and the target machine dependent and independent optimizations. These phases optimize the sEc-opcode by knowing the semantics of the JVM bytecode in the sEc-opcode. They also improve the efficacy of state-of-the-art target optimizations by feeding precise information to the optimizer, resulting in an efficient coupling of the sEc-opcode to the target machine. Along with these optimizations, the sEc-rewriting technique can switch the sEc-opcode to a more efficient implementation after runtime binding – exploiting runtime binding constants.

BrouHaHa[19], Objective Caml and Interpretation of C[12] are some implementations that make use of "macro" opcodes similar in concept to sEc-opcodes The sEc approach differs from these in the following ways:

(1) The sEc-opcode is produced offline by taking into account the dynamic characteristic, control flow and data flow of the embedded Java application.
(2) The cost of dispatch for a RISC-like opcode set is relatively greater (compared to the cost of executing the bytecode) than it is for Java bytecode. On the other hand the Java bytecode is semantically rich – most of the bytecodes can be decomposed into sub-operations and optimized. This demands a complex analysis to optimize sEc-opcodes and is done offline instead of at runtime.
(3) The sEc-opcode optimization exploits runtime binding constants as discussed in the sEc-rewriting section.

Superoperators[12] for ANSI C interpreters are a technique for specializing a bytecoded C interpreter according to the program that it is to execute. Superoperators make use of an lcc tree-based IR to synthesize superoperators. The sEc-opcode differs with respect to superoperators as follows. The semantic content of the lcc IR is very much the same as real machine instructions [9] – it consists of expression trees over a simple 109-operator language – hence folding of instruction using tree pattern[11] matching works every well. On the other hand, lcc's tree structure limits the effectiveness of this system. Putting it in another way, the lcc IR is the sequence of trees *at the semantic level of the target machine.* In contrast, the Java bytecodes are semantically rich– each bytecode is composed of many sub-operations - because of the high level of abstraction. The larger the sEc-opcode context, the more JVM dependent and independent optimization can be done – e.g. elimination of stack operands, redundant local variable access, elimination of redundant sub-operation in sEc-code bytecode sequence like method attribute check, null value check etc. After the JVM related optimizations, target machine optimizations are done by a compiler like GCC (-o). In fact gcc's RTL, (Register Transfer Language [10], the intermediate form used for most of the optimizations and for code generation in GCC can be considered as a counterpart of the lcc IR. The Superoperator technique is integrated into the lcc compilation and uses the lcc backend, which does not do advanced global optimization.

Just In Time compilation (JIT): The principle of the JIT compilation is, in general, to dynamically compile a method to native code – after some threshold number of calls to the method has occurred - pausing the application execution. Extensions and refinements of the JIT principle have spun-off many techniques under the following constraints:

1. Generating efficient native code: Optimized native code is generated on the fly by adapting traditional optimization techniques to run-time code generation.
2. Efficient code generation techniques: Optimize the JIT techniques themselves for size and speed, executing efficient code generation and register allocation algorithms. Some of recent studies in this area are Hotspot[6] CACAO[34][35] from DEC, Jalapeno[28] (now called the "Jikes RVM") from IBM, Annotated JVM[31] and Hybrid JIT [3]

Native Java Compilers: Unlike with JIT compilers Java source and binary (classes) are statically compiled to native code. This method does not have constraint 2 of JITs stated above, and hence better native code can be generated by employing state-of-art optimizations. Generally these methods disallow dynamic class loading. Toba[29], Harissa[30] and commercial products like TowerJ™and Cygnus (GNU) Java native compiler fall into this category.

7 Conclusions

Semantically enriching the Java bytecode using the sEc technique is an innovative JVM optimization technique. The sEc Technique avoids the problem of efficiently integrating traditional compilation phases into the Java runtime environment, which is a widespread problem for JIT techniques.

We have presented an abstract model for a portable way of determining the sEc-opcodes and an efficient code generation. The sEc-opcode optimization is classified into a 5-phase process, a JVM dependent, JVM independent, sEc-rewriting, target dependent and target independent. This model shows that a traditional compilation optimization problem exists with equal complexity in the JVM dependent and independent optimization phases of sEc-opcodes. The five phases of the optimization make the sEc-opcode implementation efficient by tightly coupling it to the target machine. This has shown that more than opcode dispatch optimization can be achieved by employing JVM independent and dependent optimizations, sEc-rewriting and the aggressive target machine related optimizations.

We have shown in our preliminary exploration of the implementation that the sEc technique can increase the speed of JVM interpretation by orders of magnitude for some embedded Java applications. The sEc technique employs 'C' as its intermediate language, and hence is portable to many embedded platforms, which reduces porting and retargeting cost and time. We believe that the speedup gain from the sEc technique in its aggressive form will be comparable to the JIT technique. We have also shown that the sEc technique provides fine-grained tradeoff of speed and space in an embedded JVM. More information about the sEc Technique can be found in the Hewlett-Packard laboratories Technical Report[2].

As next steps, we intend to evaluate quantitatively the different optimizations discussed in this paper. We also want to apply the sEc technique to dynamically loaded modules by *'probabilistic based sEc-opcode deduction'* for statically determinable dynamic classes. We are also interested in quantitatively evaluating the foot print overhead of sEc-technique.

8 Acknowledgments

Mr Sam Midkiff deserves a special thanks for helping us to improve the readability of this paper. We specially acknowledge Nagarajan K and Joseph Koshy for the technical discussions and encouragement we got from them. We also thank members of our research JVM team Devaraj Das and Venkata Ramani.

We thank Subrahmanyam V. S., Jeff Morgan and Ananthraman P. N. for their enduring support through out the execution of the project. Finally, we express our appreciation to the anonymous reviewers for their constructive comments and suggestions.

9 References

[1] *Coorg: Design of a Virtual Machine for Java on Embedded Systems, Hewlett-Packard Laboratories – Technical Report* - HPL-2001-68

[2] *sEc: An Interpreter Optimization technique for Embedded Java Virtual Machine, Hewlett-*

Packard Laboratories Technical Report - HPL-2001-69

[3] *A Hybrid Just*-In-Time Compiler That Consumes Minimal Resources, Geetha Manjunath, Hewlett-Packard, US patent number 6332216, issued on 18-Dec-01.

[4] Java ™ Virtual Machine Specification, Tim Lindholm and Frank Yellin, *The Java Series, Addison Wesley Publications, ISBN 0-201-63452-X.*

[5] The Java Language Specification, James Gosling, Bill Joy and Guy Steele, *Addison Wesley Publications.*

[6] The Java Hotspot™Performance Engine Architecture. *Javasoft*

[7] Compilers - Principles, Techniques, and Tools, Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, - *Addison Wesley Publications, ISBN 0-201-10194-7*

[8] Computer Architecture A Quantitative Approach, David A. Patterson, John L. Hennesy – Morgan Kaufmann Publishers, Inc. ISBN 1-55860-069-8

[9] A Code Generation Interface for ANSI C, Christopher W. Fraser, David R. Hanson, *Software – Practice And Experience, Vol. 21(9), 963-988 (September 1991)*

[10] Using and Porting GNU CC, Richard M. Stallman, *Free Software Foundation, Cambridge, MA, 1990.*

[11] Code Generation Using Tree Matching and Dynamic Programming, A. V. Aho, M. Ganapathi, S. W. K. Tjiang –*ACM Transaction on Programming Languages and Systems, October 1989.*

[12] Optimizing an ANSI C Interpreter with Superoperators, Todd A. Proebsting, *Proc POPL'95 pages 322-332*

[13] The Annotated C++ Reference Manual, Marget A Ellis and Bjarne Stroustrup, *Addison Wesley Publications.*

[14] Inside the C++ Object Model, Stanley B. Lippman, *Addison-Wesley Publishing.*

[15] Simple and Effective Analysis of Statically-Typed Object-Oriented Programs, Amer Diwan, J. Eliot B. Moss, Kathryn S. McKinley, OOPSLA 96: *Eleventh Annual Conference on Object-Oriented Programming Systems, Languages , and Appliccations.*

[16] Byte Code Engineering with the JavaClass API – Markus Dahm, Berlin, Technical Report B-17-98,

[17] JVM Subsetting for an Embedded Application, Devaraj Das, Geetha Manjunath, Internal Technical Report – HPLabs

[18] Optimizing direct threaded code by selective inlining, Ian Piumarta and Fabio Riccardi,, *ACM SIGPLAN 1998*

[19] BrouHaha – A Portable Smalltalk Interpreter - Eliot Miranda, Proc. OOPSLA 1987. *Published as SIGPLAN Notices 22(12):354-365.*

[20] A Portable Forth Engine, M. Anton Ertl, *Proc, euroForth 1993,*

[21] Compilers and Computer Architecture - William A. Wulf, Carnegie-Mellon University, IEEE Computer Journal, July 1981.

[22] A Tree-Based Alternative to Java Byte-Codes – Thomas Kistler and Michael Franz, *University of California, Irvine.*

[23] Abstract Interpretation : A Unified Lattice Model For Static Analysis Of Programs by Construction Or Approximation Of Fixpoints – Patrick Cousot and Radhia Cousot, Fourth ACM symposium on Principles of Programming Languages, 1977.

[24] Reducing garbage in Java – C. E. McDowell, http://www.cse.ucsc.edu/research/embedded/pubs/gc/index.html

[25] The Stack Allocation Optimization – Real Time Java discussion group, http://www.nist.gov/itl/div896/emaildir/rt-j/threads.html

[26] Garbage collection can be faster than stack allocation - Andrew W. Appel. *Information Processing Letters 25(4):275-279, 17 June 1987.*

[27] Optimal code generation for expression trees – A. V. Aho and S. C. Johson, *Journal of the ACM 23(3): 488 – 501, July 1976.*

[28] The Jalapeno Dynamic Optimizing Compiler for Java – Michael G. Burke, Vivek Sarkar, J. Cho, M. J. Serrano, S. Fink, V. C. Sreedhar, David Grove, Michael Hind, H. Srinivasan. – *JAVA'99 ACM.*

[29] Toba: Java For Applications, A Way Ahead of Time (WAT) Compiler – Todd. A. Proebsting, J. H. Hartman, G. Townsend, Tim Newsham, Patrick Bridges, S. A. Watterson. – *The University of Arizona*

[30] Harissa: A Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code – Gilles Muller, Barbara Moura, Fabrice Bellard, Charles Consel – *IRIA / INRIA – University of Rennes.*

[31] An Annotation-aware Java Virtual Machine Implementation – Ana Azevedo, Alex Nicoloau, - University of California, Irvine Joe Hummel – University of Illinois, Chicago.

[32] DashO-Pro – preEmtive solutions http://www.preemptive.com

[33] A Comparison of TowerJ™ and HotSpot™ Implications for Enterprise Java Deployment[White paper] – Tower Technology Corporation – http://www.towerj.com/products/whitepapertjhs.shtml.

[34] Efficient Java VM Just-in-Time Compilation – Andreas Krall, PACT'98

[35] CACAO – A 64 bit JavaVM Just-in-Time Compiler – Andreas Krall and Reinhard Grafl.

JaRTS: A Portable Implementation of Real-Time Core Extensions for Java

Urs Gleim
Siemens AG, Corporate Technology
urs.gleim@mchp.siemens.de, http://www.siemens.com/ct/

Abstract

To implement real-time tasks in Java the Java Virtual Machine has to be enhanced by the ability to run threads in a predictable time and to access hardware directly. One approach is to provide a real-time add-on for standard Java Virtual Machines. Benefits of this approach are a better predictability, a better scalability and more flexibility.

In this paper JaRTS—an implementation of the *Real-Time Core Extensions* specification—is presented. Some implementation details are inspected and first results are shown.

1 Introduction

Complexity of embedded and real-time systems increases from year to year and can only be handled by providing state-of-the-art programming languages. Usually software for those systems is written in C and even assembly language. This implies in-depth knowledge of the underlying hardware and the real-time operating system (RTOS) used. To handle complex software effectively a high abstraction layer of the programming language is needed. The Java Language provides an appropriate abstraction layer but is not designed for resource limited embedded systems and systems with real-time requirements. Thus the *Java Language Specification* [1] and the *Java Virtual Machine Specification* [2] need to be enhanced. In order to add real-time capabilities this is done by several specifications. Implementations of these specifications were not available until December 2001 when TimeSys [13] released the first reference implementation of the Sun's *Real-Time Specification for Java* (RTSJ) [3]. The *Real-Time Core Extensions* (RTCE) [4] are a competing specification by the Real-time Java Working Group.

In this paper we focus on real-time software such as control software for industrial automation. In this area there are systems running real-time software as well as non-real-time parts like graphical user interfaces on the same machine (e.g. the SICOMP industrial microcomputers [15]). On the other hand there are small controllers with very limited resources. Since RTCE fits better for these systems—this will be explained later—we implemented RTCE instead of RTSJ.

Designing real-time systems requires an overall system view. You cannot look at single parts like the Java Virtual Machine separately (assuming a common single processor system is used; hardware related topics are excluded). Therefore the paper starts with a definition of real-time systems, which are targeted. Based on that, current real-time operating system architectures are described in a nutshell. After an overview over the current real-time Java approaches the concept of the JaRTS Java real-time Java compiler will be discussed. Since real-time tasks run in a separate runtime environment the communication between real-time and non-real-time parts is shown in detail. Finally benchmarking results comparing JaRTS to other solutions are presented before we conclude.

2 Real-time systems

The term "real-time" is often used in different contexts with a different meaning. For example online systems with short response times are often named "real-time" systems. This is not meant in this paper. A canonical definition of a real-time system from Donald Gillies [12] is the following:

> A real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time at which the result is produced. If the timing constraints of the system are not met, system failure is said to have occurred.

If the timing constraints *must always* be met we speak of a *hard real-time system*, it would be *soft real-time* if missed deadlines only lead to less throughput or an acceptable reduced quality of service. Most real-time systems have additional requirements concerning robustness and availability, which are not discussed in this paper.

Writing hard real-time applications for single-processor systems on top of an multi-threaded operating system needs in-depth knowledge of the scheduling strategy and the worst case times needed for system calls in order to ensure the timeliness of the application. Furthermore worst-case preemption times of every thread and especially system calls have to be known. This leads to one important parameter, the interrupt latency. *Interrupt latency* is the time between occurrence of the hardware interrupt and the entry of the software interrupt handler (interrupt service routine, ISR).

3 Real-time operating systems

The difference between non-real-time operating systems like Windows and Linux and real-time operating systems like VxWorks is that real-time operating systems have short guaranteed thread preemption times and interrupt latencies. A standard Linux for example has interrupt latencies up to a few hundred milliseconds [11] while the latencies of current real-time operating systems are around tens of microseconds which are guaranteed by the system architecture. Worst-case times of a standard Linux can be determined by measurements but there is no guarantee that there are worse cases with even longer latencies.

Real-time operating systems are much more uncomfortable to program than desktop operating systems. They often do not provide a process concept (memory protection), which separates memory spaces of different applications and there is not much abstraction from the underlying hardware. Because applications running on embedded devices become more and more complex additional abstractions are needed. Mostly it is only a small part of a real-time application really having real-time requirements. For the rest of the software one would like to have the convenience of desktop operating systems. Therefore it makes sense to modify desktop operating systems to satisfy embedded and real time needs (or having a programming environment like Java, which we discuss later). To add real-time capabilities to a non-real-time operating system there are two approaches:

- **Preemption Improvement:** Modify the operating system to be preemptible in a defined, short time.
- **Interrupt Abstraction:** is a two kernel model running the unmodified operating system as the idle-task on top of a separate scheduler (also known as *Interrupt Isolation* or *Interrupt Virtualization*).

The fist approach is implemented for example by MontaVista [14], who improved the preemption times of Linux. Two kernel solutions for Linux are RTLinux [7] and RTAI [8]. VxWin [16] and VenturCom's RTX [17] are two-kernel solutions for Windows NT.

Due to the complexity of operating systems like Linux it is a very difficult task to improve preemption times. Because it is really hard to examine every code path in the kernel, worst case preemption times cannot be guaranteed. The interrupt abstraction model allows predicting worse case times in the small real-time layer exactly. A detailed comparison of the two approaches for Linux can be read in the articles [9] [10] [11].

4 Requirements to an implementation

The range of application for real-time Java is quite huge. It can be used in small micro controller systems as well as in large systems having the power of current personal computers and workstations. The most important requirements to a real-time Java implementation in this context are portability and scalability:

- **Portability:** The real-time Java implementation should be available for many hardware platforms and the porting effort has to be minimal.
- **Scalability:** The runtime environment should be used for small resource limited systems up to large systems with graphical user interfaces.

Portability of the Java applications is not explicitly required. It is a Java intrinsic feature that programs are easy to port to different systems. But for the mentioned systems it is necessary to access the hardware directly (access hardware registers, install interrupt handlers, ...). There are frameworks like the *Real-Time Data access* [5] targeting this issue.

5 Real-time Java approaches

Currently there are two leading Specifications adding real-time capabilities to Java. Firstly the *Real-Time Specification* (RTSJ) [3] for Java produced by the Real-Time for Java Expert Group under the auspices of the Java Community Process [18]. In December 2001 the first reference implementation for the Real-Time Specification has been released by TimeSys [13]. Secondly the *Real-Time Core Extensions* (RTCE) [4] produced by the Real-Time Java Working Group supported by HP, Microsoft and other corporations. Both specifications cover the necessary enhancements to enable Java for real-time tasks:

- **Thread scheduling and synchronization:** The Java Language Specification [1] does not define the thread scheduling exactly. In addition the ten priorities provided by Java are not enough for most real-time tasks.
- **Memory management:** Automatic memory management of Java mostly leads to unpredictable timely behavior. There is no definition of worst-case memory allocation times and even concurrent garbage collectors are not preemptible without latencies.
- **Asynchrony:** Hardware interrupts and software events can occur asynchronously in real-time systems and require an *immediate* change of the control flow.
- **Hardware access:** Java does not provide direct access to hardware registers, physical memory and handling of hardware interrupts.

RTSJ implementations require modifications of the Java Virtual Machine (JVM) internals. The application developer can choose the thread scheduling algorithm and there are several strategies for memory management. The approach of RTCE is different. Instead of modifying the whole JVM RTCE can be implemented as a real-time add-on working closely with an arbitrary existing JVM.

6 Real-Time Core Extensions

For our requirements—portability and scalability—the Real-Time Core Extensions are the more suitable approach. The real-time part is small and can be implemented to be easy to port. Beyond that it can be used as a stand-alone solution as well as in combination with an off-the-shelf JVM. The separation of a non-real-time and a real-time runtime environment allows an implementation on operating systems implementing the Interrupt Abstraction approach.

According to the RTCE specification real-time parts of the application run in a runtime environment—called *Core Java*[1]—separated from the standard Java runtime environment—called *Baseline Java* (figure 1). The Core part can also be used as a stand-alone runtime environment. It has its own set of class libraries (`org.rtwg.*`), which are also separated from the standard Java class hierarchy. Root of the Core class tree is not `java.lang.Object` but `org.rtjwg.CoreObject`. This library contains special classes for handing thread scheduling, interrupts, memory, I/O and event handling. The Core runtime environment runs with a higher priority than the Baseline Java. In so doing it is assured that the Baseline threads and garbage collection has no influence on the real-time behavior of Core threads.

The classes of the Core library are shown in figure 2. They provide only basic functionality. The standard Baseline Java can access dedicated methods of Core

[1] The term *Core Java* was not the best choice by the Real-Time Java Working Group since it has a different meaning in the standard Java community.

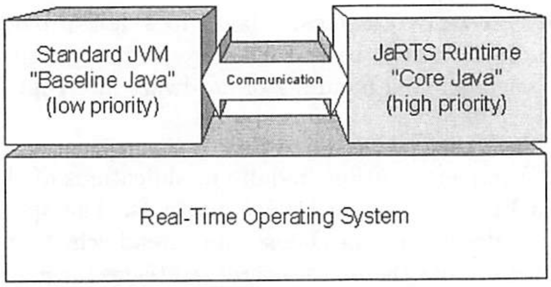


Figure 1: separation of real-time and non-real-time parts

objects, which are marked as *"Core-Baseline Methods"* (more details below). There is a small Baseline library for this communication, shown in figure 3. A detailed documentation of those libraries can be found in [4].

7 JaRTS: basic concept

For performance reasons it makes sense to compile the code to native machine code at compile time (ahead-of-time compilation). CPU time and memory is saved compared to a JIT-compiler and the costs of dynamic class loading are avoided. The real-time Java parts are used for drivers and embedded control algorithms which mostly do not need dynamic class loading (however a solution for dynamic class loading is planned, see section 14).

JaRTS (Java Real-Time by Siemens) is an implementation of a Core Java compiler and the RTCE libraries. Output of the JaRTS compiler is platform independent ANSI-C code as well as Java code for the Baseline-Core communication. The platform dependent parts are placed in separate operating system dependent include files. In addition the communication between Baseline and Core is implemented in platform dependent C files.

The prototype JaRTS compiler and runtime libraries were implemented for RTLinux. The Core parts of an application (real-time parts) are compiled to native code running directly on the real-time scheduler. For RTLinux the real-time code has to be compiled into kernel modules that can be loaded dynamically. Figure 4 shows the whole build process of a JaRTS real-time Java application. Platform dependent files for RTLinux are encircled.

```
CoreObject
      CoreThrowable
            CoreRuntimeException
            CoreException
            ScopedException
      CoreClass
      CoreArray
      AllocationContext
      SpecialAllocation
      CoreString
            DynamicCoreString
      ATCEventHandler
      ATCEvent
      CoreRegistry
      SignalingSemaphore
      CountingSemaphore
      Mutex
      Configuration
      Time
      Unsigned
      CoreTask
            ISR_Task
            SporadicTask
      IOPort

Interfaces:
      PCP
      Atomic
```

Figure 2: Core API

```
java.lang.Object
      lava.lang.Classloader
            BaselineCoreClassloader
      CoreDomain
      java.langException
      ObjectNotFoundException
      CoreBaselineRuntimeException
```

Figure 3: Baseline API

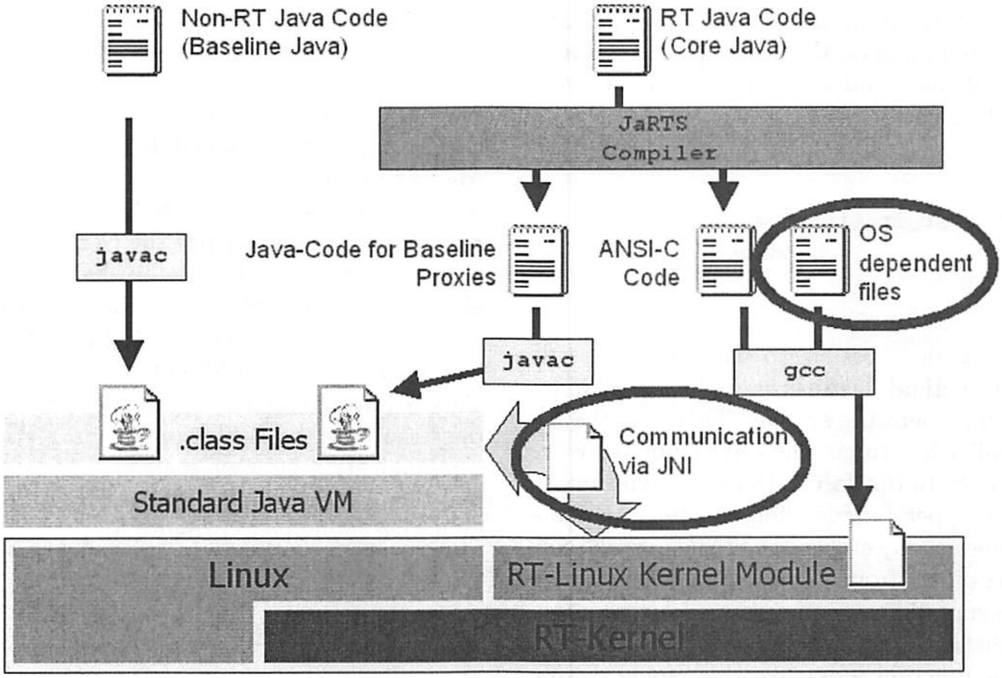


Figure 4: build process and platform dependent files

We implemented the communication of Baseline and Core Java via JNI (*Java Native Interface*) that is supported by most JVMs. Since all the real-time processing should take place in the Core part the interface between the Core and the Baseline part of an application is very small and the poor performance of JNI is not an issue.

8 JaRTS implementation

To get quick results the JaRTS compiler is based on in Open Source Java to C translator called *bock*, written by Charles Briscoe-Smith [6]. This translator did not support threads and therefore no synchronization. This was added to implement `CoreTask`, `ISRTask` and `SporadicTask` of the RTCE specification. Since we needed a periodic task—which is a task executing periodically after a specified time—we added an additional class `PeriodicTask`.

The Core library was implemented straightforward, mapping mutexes, interrupt handlers etc. directly to the functions provided by the underlying system. Native C code used in the libraries is weaved directly into the output of the compiler. So there is no additional runtime overhead for invoking native methods.

In the following section handling of the interrupt service routines, the communication between real-time and non-real-time parts as well as the memory management strategy is described.

9 Interrupt service routines

Because of object orientation Java requires runtime overhead even for conceptually simple tasks like interrupt handling. An interrupt service routine (ISR) in Java (RTCE) is implemented similar to a thread. A `work()` method has to be implemented in a subclass of `ISRTask`. The interrupt number is a member variable of this class. There can be several instances of the same class handling different interrupts.

To access member variables in the C translation of the `ISRTask` a pointer to the corresponding object has to be known inside the translated `work()` method.

In real-time operating systems ISRs are assigned to a hardware interrupt by a system call. This call has only two parameters: the interrupt number and the address of the handler method. In RTLinux it would look like this:

```
rtl_request_irq(irnumber,
                handler);
```

Of course it is not possible to pass arguments to the handler method because it will be called by the underlying operating system. Therefore the interrupt handler has to get the object pointer from somewhere else. In the JaRTS runtime environment there is a wrapper function for all interrupt handlers fetching an object pointer of the corresponding ISRTask object from a small table whenever an interrupt occurs. This wrapper is used for all interrupts handled by Core Java applications. Roughly the wrapper function looks like this (some details left out):

```
void isr_entry(int irq) {
  isr_object* o=table[irq];
  (o->methods->work)(o);
}
```

This is connected to every handled interrupt by

```
rtl_request_irq(irnumber,
                isr_entry);
```

The object pointers are written to the table during initialization of the ISRTask. Accessing the table (table[]) does not require mutual exclusion because it will not be reallocated and values are only read (after the initialization phase where the handler will not be called with the current interrupt number).

So the Java overhead for ISRs is the table access and an additional function call in the isr_entry() function. This leads to slightly longer interrupt latencies compared to C but this is a predictable worst-case time. This time can be calculated by looking at the (assembly language) output of the C compiler with the assumption that all data and the code of the work() method is not in cache.

10 Baseline-Core communication

The prototype implementation was done for RTLinux [7]. As mentioned, RTLinux uses Interrupt Abstraction to make Linux real-time capable. This is a two-kernel solution where Linux runs on top of a real-time scheduler. Hence the Core Runtime environment has to run directly on the real-time scheduler and communication must be possible from the real-time threads to the non-real-time threads running on a standard JVM on top of the Linux kernel. This is implemented by some communication routines that use the FIFOs provided by RTLinux for communication between real-time threads and non-real-time Linux threads. These FIFOs are accessed by native code via JNI (figure 5).

10.1 The Core part

To access methods in the Core code from Baseline these methods have to be marked as Core-Baseline methods with the keyword *baseline*[2], for example (CoreTask is the Core thread class):

```
public class Controller extends
CoreTask {
...
public void baseline setSpeed(...)
...
public void baseline getSpeed()
...
```

By now only the methods which may be accessed by Baseline Code are marked.

Objects instantiated in the Core part can be published to Baseline with a string name:

```
MyCoreObject co =
       new  MyCoreObject();
CoreRegistry.
    publish("myObject01", co);
```

Objects are published by sending the name, the type and a unique object ID via FIFO to the Baseline

[2] There is an notation that doesn't introduce a new keyword: calling CoreRegistry.registerBaseline(<mthd_signatures>) in the static initializer

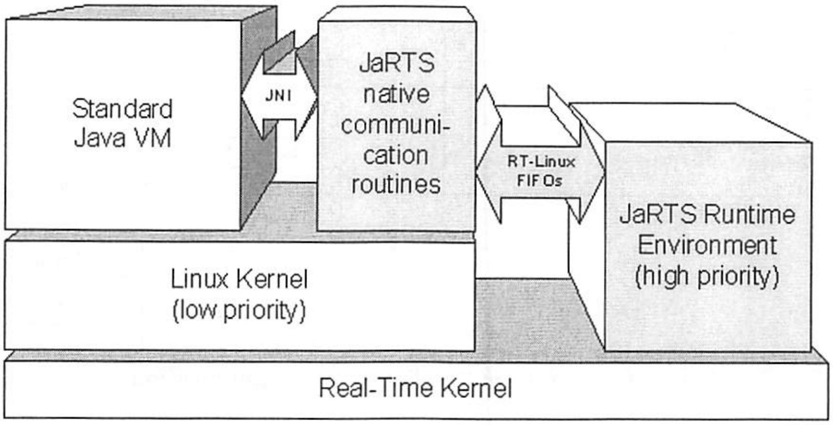

Standard Java VM
JNI
JaRTS native communication routines
RT-Linux FIFOs
JaRTS Runtime Environment (high priority)
Linux Kernel (low priority)
Real-Time Kernel

Figure 5: separation of real-time and non-real-time parts

JVM (described below). To receive method calls from Baseline a loop is listening at another FIFO for commands (`CoreListener`).

10.2 The Baseline part

The Baseline instance for Baseline-Core communication is `CoreDomain`. Previously published Core objects can be obtained by calling `CodeDomain.lookup()`, for example:

```
    MyCoreObject bco =
    CoreDomain.lookup("myObject01");
    bco.foo(42, 3.1416);
```

What `CoreDomain.lookup()` actually returns is a proxy object doing the communication with the corresponding Core object. The JaRTS compiler generates the proxy classes for each Core class containing Core-Baseline methods. These proxy classes are subclasses of a Baseline version of `CoreObject` (figure 6).

We added an internal class `BaselineCoreConnector` that handles the connection. It is a singleton and used by the proxy objects. Therefore it is a static member of the proxy superclass `CoreObject`. A `BaselineFifoListener` waits for commands like publishing objects in the Core as well as for returning functions (figure 7). The actual communication is done in native files written in C. These files have to be adapted when porting JaRTS to other platforms.

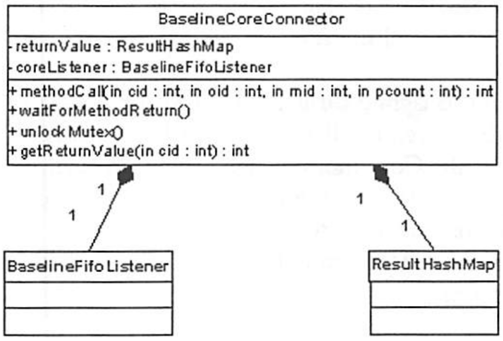

BaselineCoreConnector
- returnValue : ResultHashMap
- coreListener : BaselineFifoListener
+ methodCall(in cid : int, in oid : int, in mid : int, in pcount : int) : int
+ waitForMethodReturn()
+ unlockMutex()
+ getReturnValue(in cid : int) : int
1
1
1
1
BaselineFifoListener
ResultHashMap

Figure 7: auxiliary classes for the JaRTS implementation

10.3 Calling Core-Baseline methods

Figure 8 shows a sequence diagram of the publication and a Core-Baseline method call. Arrows crossing the horizontal line between Baseline and Core are standing for data sent through the FIFOs. The data sent is described in the balloons.

Name, type and ID (for optimization) of a published object are received by the `FifoListener` and internally stored in a hash map of `CoreDomain` for lookup. The latter is not shown in the sequence chart. For every published object a proxy object on the Baseline side is created. This proxy objects contains wrappers for all the Core-Baseline methods in the corresponding Core object. When called, these wrappers send a "call method" command followed by the object ID, method ID and the parameters to the `CoreListener`. Since multiple threads can call

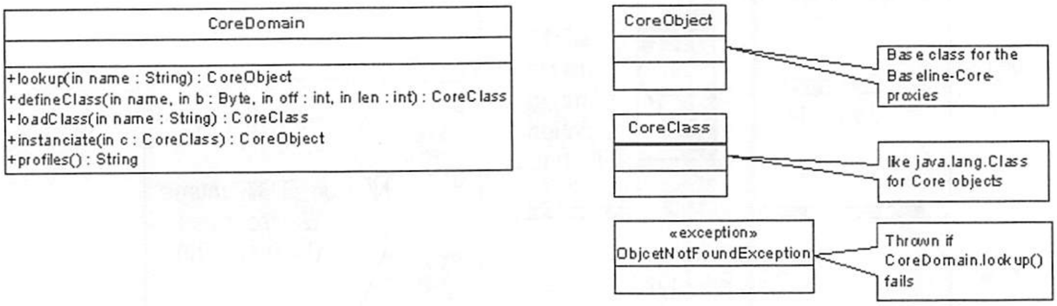

CoreDomain
+lookup(in name : String) : CoreObject
+defineClass(in name, in b : Byte, in off : int, in len : int) : CoreClass
+loadClass(in name : String) : CoreClass
+instanciate(in c : CoreClass) : CoreObject
+profiles() : String
CoreObject
Base class for the Baseline-Core proxies
CoreClass
like java.lang.Class for Core objects
«exception»
ObjcetNotFoundException
Thrown if CoreDomain.lookup() fails

Figure 6: classes for Baseline-Core communication

the same method concurrently an additional call ID is sent. This is important to distinguish the method returns. The call ID can be a thread ID or a unique number. On the Core side a thread is started executing the wanted method.

The calling Baseline thread is blocked by a `wait()`. The calling thread will be resumed by a `notify()` after the calls Core method has sent its return values. Since multiple threads can call Core-Baseline methods concurrently return values are stored in a hash map with their call ID and can be fetched by the wrappers.

11 Memory management

The JaRTS runtime environment does not support garbage collection as known from Standard JVMs. Most real-time software components like interrupt handlers and device drivers would derive no or only a little benefit from having automatic garbage collection. On the other hand garbage collection imposes significant costs in terms of runtime efficiency, predictability and system complexity.

RTCE provides the concept of *Allocation Contexts*. Objects are allocated on an Allocation Context of the current thread. The Allocation Context is released when the thread terminates or can be released explicitly by the programmer. It is also possible to allocate objects on dedicated Allocation Contexts. The second memory strategy of RTCE, allocation objects on the stack is currently not implemented.

	Sieve	*Loop*	*Logic*
JaRTS	180	227	933
CVM	60	61	60
Sun 1.3	285	1098	769

Table 1: benchmarking results

12 Results

Up to now the results are promising:

Latency periods: As expected the interrupt latency periods are nearly the same as the latencies of interrupt handlers implemented in C. The average time between a hardware interrupt (at the parallel port of an 150 MHz Pentium) is 7 µs for interrupt service routines in C. The Java version needs about 9.5 µs. The Java overhead of 2.5 µs is acceptable for most real-time systems.

Performance: Performance was tested with a sieve of Eratosthenes, which calculates prime numbers, a loop test sorting values and a simple test of logical decisions.

Table 1 shows the results (score, higher values are better). Compared to Sun's HotSpot Client VM (1.3) the code generated by JaRTS was about 20% faster for the logic test. The other tests ran about 58% (sieve test) up to 366% (loop test) slower. Compared to Sun's CVM [20] (on which the TimeSys real-time reference implementation is based) JaRTS is 3 times up to more than 15 times faster.

Improvements of the JaRTS performance should still be possible since there was no effort in optimizing performance, yet.

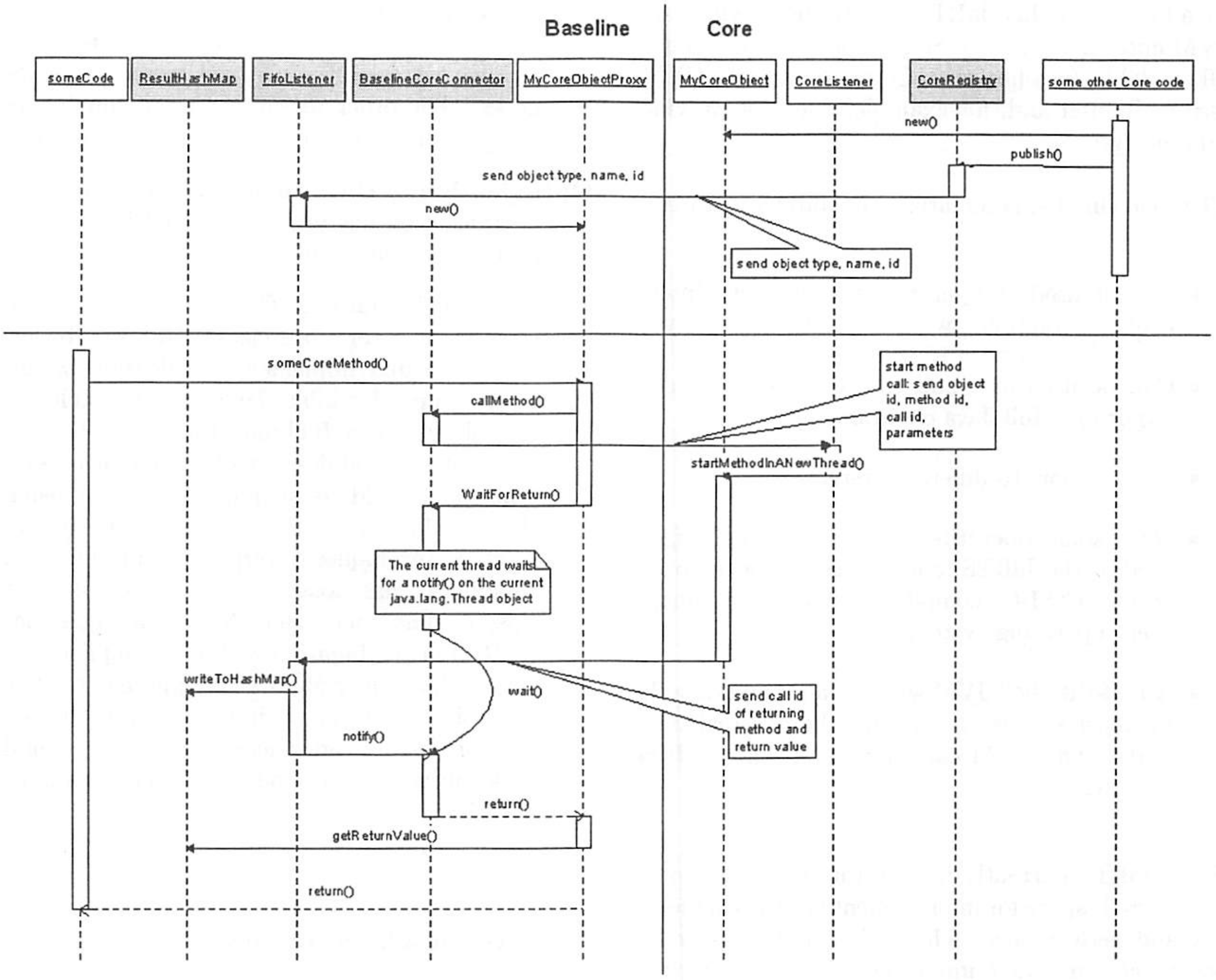

Baseline
Core
someCode
ResultHashMap
FifoListener
BaselineCoreConnector
MyCoreObjectProxy
MyCoreObject
CoreListener
CoreRegistry
some other Core code
new()
publish()
send object type, name, id
new()
send object type, name, id
someCoreMethod()
callMethod()
start method call: send object id, method id, call id, parameters
startMethodInANewThread()
WaitForReturn()
The current thread waits for a notify() on the current java.lang.Thread object
writeToHashMap()
wait()
send call id of returning method and return value
notify()
return()
getReturnValue()
return()

Figure 8: publishing methods and method call

Portability: There are only a few files containing platform dependent code. This code contains wrappers for handling threads, mutexes and semaphores and the code controlling the FIFOs for Baseline-Core communication. Because the JaRTS compiler generates platform independent ANSI-C code it can be ported to any system providing an ANSI-C compiler easily. Due to the separation of real-time and non-real-time parts it can also be run on systems using Interrupt Abstraction.

Code size: The Core run-time currently has a footprint of about 700 kilobytes including the Core libraries. This is about the same size as current MIDP [19] implementations, used for mobile phones. Until now there were no optimization efforts in this area, so there is still space for improvement.

13 Conclusion

We wanted to show that it is possible to implement a portable and scalable real-time Java extension with hard real-time capabilities.

The platform dependent code is relatively small and manageable. It is much easier to port JaRTS to a new platform than a complex JVM implementing RTSJ.

Currently the TimeSys RTSJ reference implementation is based on the CVM and the Foundation Profile libraries [20]. CVM is too big for resource limited systems (more than 2.5 MByte for CVM with Foundation Profile) where only a very small library would be sufficient.

In addition to this JaRTS can be used with any JVM implementing JNI. So the applications can use all available Java libraries. Compared to this CVM is very limited and, for example, does not provide graphics yet.

The following list summarizes the JaRTS features:

- Can be used on operating systems with Interrupt Abstraction (two kernel solution) easily
- Can be used on small embedded systems not requiring a full Java environment
- Easy to port to different systems
- Only some operating system dependent files used by the JaRTS compiler have to be adapted (rest: ANSI-C, compiler available for almost every operating system)
- An off-the-shelf JVM which is already available for most systems is used for the non-real-time parts. This JVM can support all known Java libraries.

Since JaRTS currently is a prototype implementation there is space for improvement in terms of memory and performance. The *bock* compiler was chosen to get to a first implementation very quickly. The translation was not tuned to be efficient and resource saving. Nevertheless with the current implementation it is possible to generate control programs with very low latency times and an acceptable performance.

14 Future work

As mentioned JaRTS is a prototype implementation and of course there are open issues:

Memory Management: Stack allocation has to be implemented.

Tooling: A Java runtime environment can only be used in an efficient way if there are good development tools. A remote debugger, profiling tools an a simulator for the JaRTS runtime have to be developed.

Benchmarks: A more sophisticated benchmarking suite has to be developed. One reason is to determine the performance and memory bottlenecks. The other reason are more convincing comparisons to other real-time Java solutions.

Optimization of the translation: The Java to C translation has to be optimized in terms of performance and memory consumption.

Dynamic class loading: Concepts for dynamic class loading (of Core classes) have to be investigated and implemented. Possible would be to compile loadable classes into loadable native libraries (for RTLinux this would be separate kernel modules, for other operating systems this would be shared libraries) or using the standard Java Bytcode (class files) and common techniques (interpreter, JIT compiler, compile at class-loading time). Compiling at class loading time using the existing compiler (JaRTS in combination with a C compiler) has very high memory and CPU requirements. The JIT solution needs much effort to port it to a new processor. So a feasable solution would be an interpreter or the precompiled loadable libraries.

15 Acknowledgements

I would like to thank my colleague Thomas Henties for a large part of the implementation and the *Siemens CT SE 2 Embedded Team* for many useful suggestions.

Thanks also to Charles Briscoe-Smith who wrote the *bock* compiler and published the source code in the Internet.

References

[1] James Gosling, Bill Joy, Guy Steele, Gilad Bracha: *The Java Language Specification (Second Edition).* Addison-Wesley. 2000 `<http://java.sun.com>`

[2] Tim Lindholm, Frank Yellin: *The Java Virtual Machine Specification (Second Edition).* Addison-Wesley. 1999 `<http://java.sun.com>`

[3] Greg Bollella e.al.: *The Real-Time Specification for Java (version 1.0).* Addison-Wesley. Dec. 2001
<http://www.rtj.org>

[4] J-Consortium, Real-Time Java Working Group: *Real-Time Core Extensions (revision 1.0.14).* Sept. 2000
<http://www.j-consortium.org/rtjwg/index.shtml>

[5] J-Consortium, RTDA Working Group: *Real-Time Data Access (version 1.8).* Febr. 2001
<http://rtawg.khe.siemens.de/rtawg.html>

[6] *Bock compiler.* available at:
<http://packages.debian.org/unstable/devel/bock.html>

[7] *RTLinux by FSM labs:*
<http://fsmlabs.com/community/>

[8] *RTAI (Real Time Application Interface):*
<http://www.aero.polimi.it/projects/rtai/>

[9] Tim Bird: *Comparing two approaches to real-time Linux.* Dec. 2000
<http://www.linuxdevices.com/articles/AT7005360270.html>

[10] Tim Bird: *Two Approaches to Real-Time Services in Linux.* RTC Magazine, November 2000

[11] Jim Ready, Kevin Morgan: *Application-Oriented Approach to Real-Time Linux.* RTC Magazine, November 2000

[12] *Comp.realtime: Frequently Asked Questions (FAQs).* (version 3.5)
<http://www.faqs.org/faqs/realtime-computing/faq/>

[13] *TimeSys.*
<http://www.timesys.com/>

[14] *MontaVista.*
<http://www.mvista.com/>

[15] Industrial microcomputers *SICOMP.*
<http://www1.ad.siemens.de/sicomp/index_76.shtml>

[16] *LP-Elektronik.*
<http://www.lp-elektronik.com/>

[17] *VenturCom.* Real-time Extensions for Windows NT
<http://www.vci.com/products/windows_embedded/rtx.asp>

[18] *Java Community Process.*
<http://www.jcp.org/>

[19] *Mobile Information Device Profile.*
<http://java.sun.com/ products/midp/>

[20] *CVM* and the *Foundation Profile.*
<http://java.sun.com/products/cdc/>
<http://java.sun.com/products/foundation/>

Targeting Dynamic Compilation for Embedded Environments

Michael Chen and Kunle Olukotun
Computer Systems Lab
Stanford University
mikey@hydra.stanford.edu, kunle@ogun.stanford.edu

Abstract

A generally held notion is that high quality code comes with high compilation cost. As a result, previous efforts at minimizing dynamic compilation costs have focused on designing fast, lightweight compilers that sacrifice code quality for compilation speed, and resource intensive approaches that combine multiple engines to limit expensive optimizations to critical sections. In this paper, we show one possible way fast compilers can be constructed to generate high quality code. We have implemented microJIT, a small and portable just-in-time (JIT) compiler for Java that can produce high quality code 2.5x faster than a comparable dataflow-based compiler and 30% faster than a compiler that performs only limited optimizations. We use dataflow techniques, but speed up compilation by minimizing the number of major compiler passes given the number of optimizations performed. Architectural features of our compiler also allow it to perform instruction set dependent optimizations efficiently. microJIT achieves these high compilation rates while still maintaining small static and dynamic memory requirements. This compiler can be highly effective in an embedded system where computing and memory resources are highly constrained and where multiple target platforms must be supported.

1. Introduction

Dynamic code generators differ from traditional compilers because they must carefully consider their impact on runtime performance. It is widely believed that generating optimized code comes with high compilation times. Two approaches to reducing dynamic compilation overheads have been popularized.

The first approach uses techniques that focus solely on generating code quickly [1][9]. This usually involves focusing only on simple optimizations, or relying on imprecise analysis or heuristics to speed up compilation. This helps alleviate compilation times, but penalizes long-run performance through poor code quality.

The second approach is through lazy compilation. A lazy compilation system usually combines a fast, non-optimal compiler or interpreter with an expensive compiler [10]. The key goal of this system is to minimize expensive optimizations. Runtime profiling selects which portions of code to interpret and which portions to recompile and optimize aggressively.

In this paper, we show the gap between high quality code and fast compilation may be bridged a third way: by improving the compilation performance of the dynamic compiler. We have implemented microJIT, a small and portable dataflow compiler for Java that produces optimized code 2.5x-10x faster than comparable dataflow-based compilers and 30% faster than a compiler that performs limited optimizations. These compilation rates are achieved while maintaining small static and dynamic memory requirements. The major contributions of this paper are:

- Architecture and implementation of a fast, portable, and small optimizing compiler. Compilation speedup was achieved by minimizing major compiler passes for the number of optimizations performed: global and local code optimizations are applied as intermediate expressions are generated, registers are allocated concurrent with code generation, and dataflow information is efficiently communicated between compiler passes. Our compiler also implements an extension infrastructure to help support fast machine dependent optimizations.

- Experimental results that show using this fast compiler alone can compete against systems using interpreters, lazy compilation, and fast compilers on long and short run applications, with less total system cost.

Our compiler can be highly effective in embedded systems like PDAs, tablet PCs, and thin clients targeted by Sun's J2ME (Java 2 Micro Edition) platform, using the CDC and CVM configuration (http://java.sun.com/j2me/). These environments have highly constrained computing (30Mhz – 200MHz) and memory resources (2MB – 32MB RAM and ROM) relative to modern desktop machines, and must support many target platforms. The current standard for these devices is to interpret bytecodes, which results in at least 10x slowdown relative to native code. Our fast compiler may alleviate tradeoffs that would otherwise need to be made in this environment. Overheads for expensive code optimization can be significant for slower devices, and a fast, poorly optimizing compiler ultimately sacrifices long run performance for code generation time. Alternatively, using multiple compilation and/or execution engines required for lazy compilation add to code ROM size and require more development effort to target different machine platforms.

The rest of our paper is organized as follows. We describe related dynamic compilers and fast compilation techniques in Section 2. We outline the major optimizations performed by our compiler in Section 3 and discuss how they relate to traditional dataflow implementations in Section 4. In Section 5, we compare compilation times and code performance with other dynamic compilers. Finally, we present our conclusions in Section 0.

2. Related Work

The Sun Hotspot Java virtual machine (JVM) is the most widely known system that implements lazy compilation [10], although similar research and commercial systems by Intel, IBM and SNU exist [5][19][16]. We compare the performance of these compilers to microJIT in Sections 4 and 5. An interesting extension of lazy compilation is adaptive optimization, implemented in the Jalapeño VM [2]. Written mostly in Java, this VM moves the Java / non-Java boundary below the VM, providing more opportunities for optimization.

A critical component of fast dynamic compilation is fast register allocation. The Intel JIT compiler uses lazy code selection in the context of on-the-fly register allocation to speed up compilation [1]. Previously encountered bytecode sequences that generated the current value in a register are cached, as well as possible bytecode aliases, so that future equivalent sequences can simply be replaced by an instantiation of the register. In linear scan register allocation [17][4], a prepass computes lifetimes and lifetime holes and then directs global allocation of registers with a simple linear sweep over the program being compiled. The LaTTe JIT compiler uses another potentially effective fast register allocation scheme that requires a prepass and two sweeps [19]. In Section 4.2, we discuss how these allocators relate to the one implemented in microJIT.

There are several well-known research projects that use dynamic compilation. The SimOS project includes the Embra execution engine that uses dynamic translation to speed up architecture simulations [18]. `C ("tick-c") implements language extensions to C to support dynamic compilation of critical sections of code [14][7][13]. The Digital FX!32 project uses dynamic translation to run x86 binaries on the Alpha architecture [15]. The Dynamo project optimizes and recompiles binaries according to statistics collected from runtime profiling [3].

3. Compiler Architecture

3.1 Overview

Our compiler only makes three major passes over

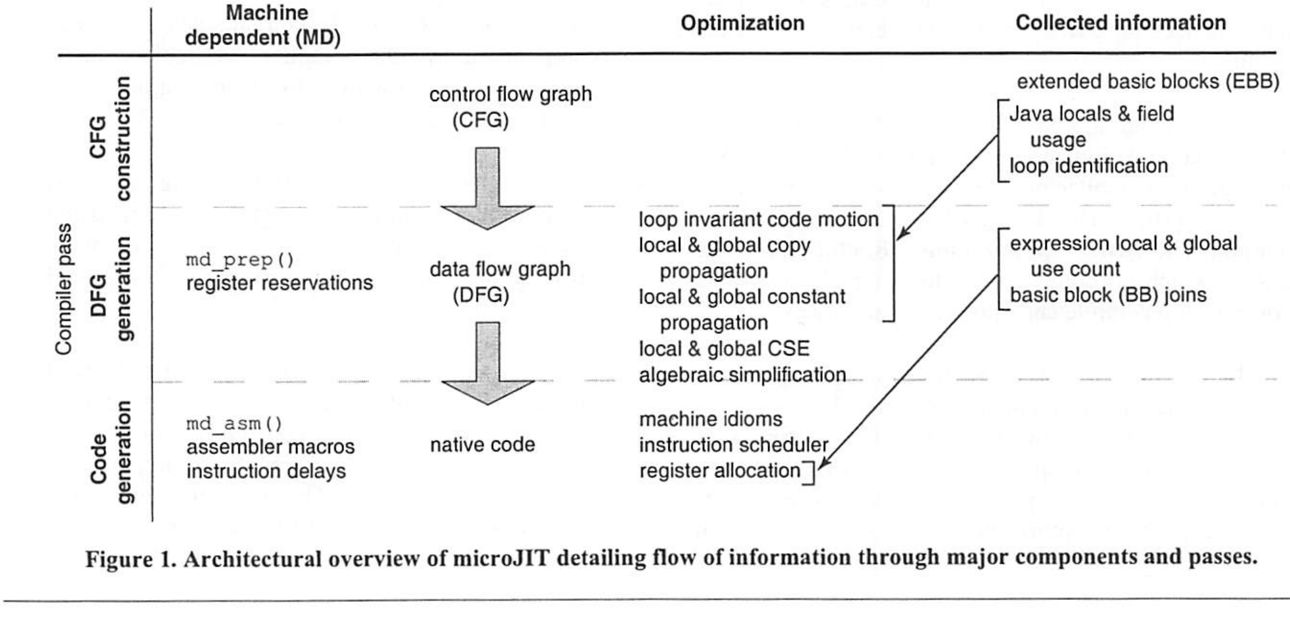


Figure 1. Architectural overview of microJIT detailing flow of information through major components and passes.

the code: a fast scanning pass to build the control flow graph (CFG), a major pass to generate the data flow graph (DFG), and then the final pass to generate the code. We found that organization represents a minimal pass configuration that gathers enough good information for the following pass to compile effectively. Figure 1 graphically illustrates the high-level architecture of our dynamic compiler, showing specific work done by each pass, ISA specific components, and flow of information between the major blocks. The following sections (3.2 – 3.5) detail the major compiler passes, and steps we take to minimize compilation costs without sacrificing generated code quality.

3.2 CFG Construction

We generate a CFG of basic blocks (BB) from a single pass of the bytecode. In our compiler, a method starts as one block representing all the bytecodes. Blocks are split as branches and associated targets are found. During a block split, appropriate control flow arcs between blocks are added and adjusted. In this fashion, we can generate the CFG without touching a bytecode more than once.

This scan executes very quickly. Most bytecodes, except for control flow instructions, are not decoded during this scan, and most next pc and stack pointer offsets can be found simply by indexing the bytecode into a static lookup table. At the end of CFG construction, arcs are also added to any defined exception handlers for the BB.

From the CFG, we compute extended basic blocks (EBB) and dominator blocks using standard algorithms [12]. An EBB is a maximal sequence of BBs with one entry and possible multiple exits. Subsequent passes operate on EBBs in order to allow local optimizations to be applied to the largest possible region.

Table 1. Example IR expressions.

Class	Example expression	bytecode equivalent(s)	Arguments
Load/ store	load	`getfield` `getstatic`	1 "
Unary op	`not`	`not`	1
Binary op	`add`	`add`	2

Branches	`cbr_eq`	`ifeq`
		`if_icmpeq`
		`if_acmpeq`
			2 + target
			"
			"

Auxiliary	`call`
	
	`prologue`
		`invokevirtual`
		`invokestatic`
		`invokespecial`
		`none`
			variable
			"
			"
			0

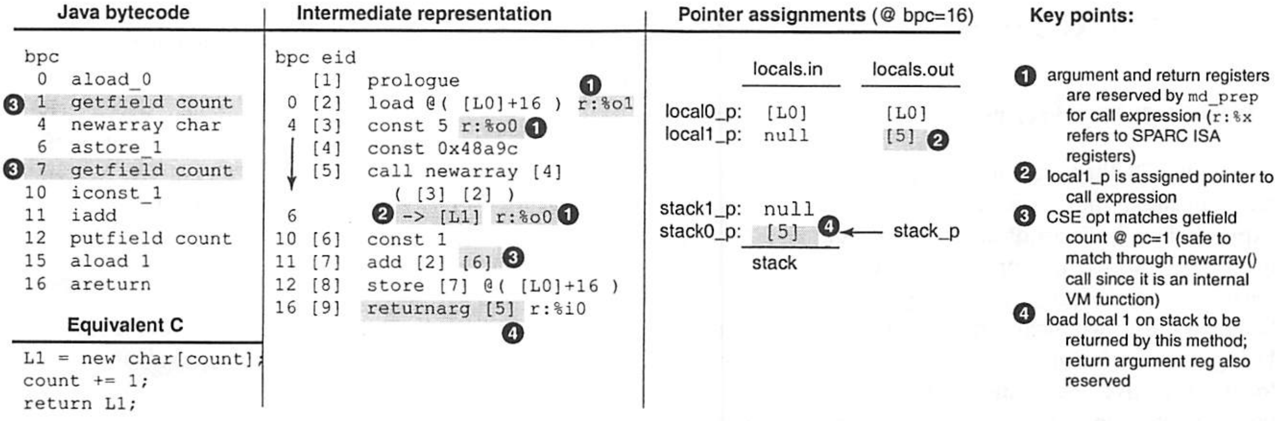
Dominator block information is immediately used to detect loops. Since `goto` statements with arbitrary target labels are not allowed in the Java programming language [8] (although it is not formally excluded from the VM specification [11]), we can expect all loops to have one entry point and be found with a natural loop detector [12].

Basic load/store statistics on local (e.g. `iload, astore`) and field (e.g. `getfield, putstatic, iaload, aastore`) accesses are recorded for each BB during CFG construction. The local and field accesses for each BB in a loop are then merged to compute their definitions and uses within the loop. This is used in the next pass for global optimizations.

3.3 DFG Generation

3.3.1 Intermediate Representation

Within BBs, we use triples to represent IR expressions. Triples are similar to quadruples used by most compilers, except that results are not named explicitly [12]. In our implementation of triples, pointers are used to refer to source argument



Java bytecode

```
  bpc
   0  aload_0
❸  1  getfield count
   4  newarray char
   6  astore_1
❸  7  getfield count
  10  iconst_1
  11  iadd
  12  putfield count
  15  aload 1
  16  areturn
```

Equivalent C

```
L1 = new char[count];
count += 1;
return L1;
```

Intermediate representation

```
bpc eid
    [1]  prologue                ❶
 0  [2]  load @( [L0]+16 ) r:%o1
 4  [3]  const 5 r:%o0 ❶
 |  [4]  const 0x48a9c
 ↓  [5]  call newarray [4]
            ( [3] [2] )
 6         ❷ -> [L1] r:%o0 ❶
10  [6]  const 1
11  [7]  add [2] [6] ❸
12  [8]  store [7] @( [L0]+16 )
16  [9]  returnarg [5] r:%i0
                   ❹
```

Pointer assignments (@ bpc=16)

```
            locals.in    locals.out
local0_p:   [L0]         [L0]
local1_p:   null         [5] ❷

stack1_p:   null
stack0_p:   [5] ❹←—  stack_p
            stack
```

Key points:

❶ argument and return registers are reserved by `md_prep` for call expression (`r:%x` refers to SPARC ISA registers)

❷ local1_p is assigned pointer to call expression

❸ CSE opt matches getfield count @ pc=1 (safe to match through newarray() call since it is an internal VM function)

❹ load local 1 on stack to be returned by this method; return argument reg also reserved

Figure 2. Example showing how bytecodes are translated to our intermediate representation

expressions. Basic expression classes of our IR are listed in Table 1. Our IR expressions more closely resemble basic machine instructions than Java bytecodes to make mapping to machine code more straightforward. Constants are represented as individual expressions so they can be properly manipulated on the Java stack as bytecodes are processed. Because we implement triples, expressions have no explicit destination.

3.3.2 Local Optimization

Our compiler is designed to perform local optimizations quickly. This is important because bytecodes sequences often reoccur, a side effect of using a stack to hold temporaries, and of complex bytecodes such as array accesses that hide repeated computations.

```
EXPR_add( expr * e, basic_block * bb )
{
    // CONSTANT OPTIMIZATION
    if( constant_arguments( e ) ){
        e->const_opt();
    }
    // ALGEBRAIC SIMPLICIATION
    else{
        e->algebraic_opt();
    }
    // LOOP OPTIMIZATION
    if( in_loop ){
        (success, match_e) = bb->loop_opt( e );
        if( success ){
            return match_e;
        }
    }
    // CONSTANT SUBEXPRESSION ELIMINATION
    (success, match_e) = bb->cse_opt( e );
    if( success ){
        return match_e;
    }
    else{
        (success, match_e)
            = bb->global_cse_opt( e );
        if( success ){
            return match_e;
        }
    }
    // SETUP MACHINE DEPENDENT INFO
    e->md_prep();
    // ADD TO BASIC BLOCK
    bb->add( e );
}
```

Figure 3. EXPR_add() pseudo code (simplified code).

Figure 2 illustrates an example of how bytecodes are translated into expressions in our IR. A local expression pointer array and a stack pointer array are maintained as we interpret bytecodes in a BB. Java locals and stack assignments simply move expressions between these two arrays and update the stack pointer. Using pointer assignments to mimic the VM stack and locals effective performs copy propagation as no intermediate copy expressions are generated for these operations. Expressions that are assigned to Java locals on BB entry and exit are listed in expressions pointer arrays for each BB (`locals.in[]` and `locals.out[]`).

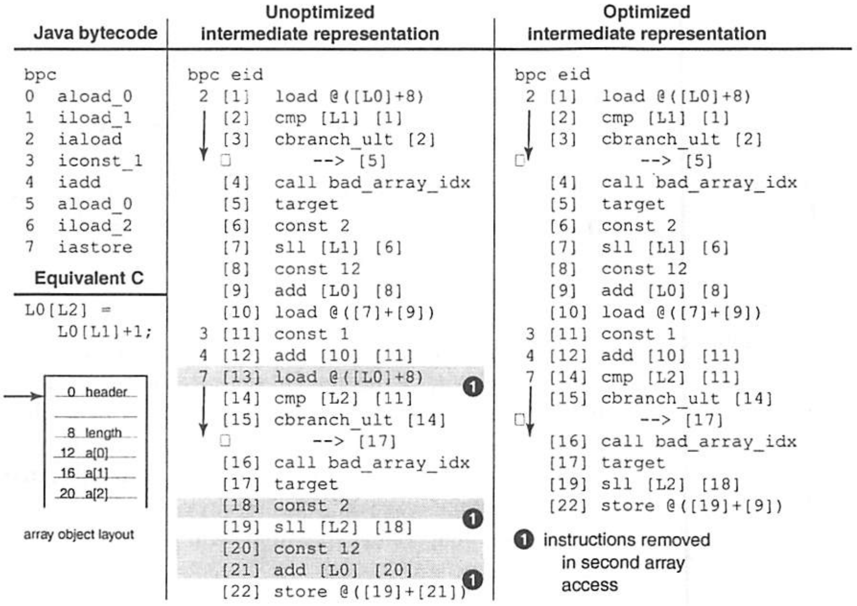
Only when we actually encounter a true operation is an expression generated for it. When an expression is created, it is submitted to the function `EXPR_add()` (see Figure 3), which checks the expression for possible optimizations before adding it to the expression list for the BB. Basic optimizations performed include constant propagation, and algebraic simplifications and reductions. Assuming a non-constant expression, local common-subexpression elimination (CSE) is applied to the new expression. Our local CSE is implemented as a backward search within the EBB for an available matching expression, with expressions hashed by operation to eliminate searching through expressions that cannot possibly match the new expression.

Most bytecodes that perform simple operations simply map to a single corresponding IR expression. Complex bytecodes, like array accesses, branches and method calls, are decomposed into IR expressions that are more representative of the instructions that must be executed on the underlying hardware. Consider the array access sequence shown in Figure 4. Decomposing allows us to optimize an array bounds check with another access to the same array, or allows us to use the same index computation for access to a different array.

3.3.3 Inlining and Specialization

Our inliner supports fast inlining of small methods (< 20 bytes) rather than full, and potentially more expensive, integration of a callee method into a caller method. Expressions in the inlined method are added to the caller context, but maintain a separate environment.

Small methods represent the most common opportunities to inline, and result in big performance gains by eliminating large calling overheads relative to work performed within the inlined call. Our inliner handles nested inlining (e.g. for optimizing subclassed object constructors like `class.<init>`), and specialization of virtual and interface methods (e.g. for optimizing object accessor methods). The inliner is also responsible for inlining fast, common case handlers for the `checkcast` and `instanceof` bytecodes, which must execute a costly class hierarchy search if the object class is not equivalent to the requested class.



Java bytecode

```
bpc
0  aload_0
1  iload_1
2  iaload
3  iconst_1
4  iadd
5  aload_0
6  iload_2
7  iastore
```

Equivalent C

```
L0[L2] =
   L0[L1]+1;
```

Unoptimized intermediate representation

```
bpc eid
 2 [1]  load @([L0]+8)
   [2]  cmp [L1] [1]
   [3]  cbranch_ult [2]
   □        --> [5]
   [4]  call bad_array_idx
   [5]  target
   [6]  const 2
   [7]  sll [L1] [6]
   [8]  const 12
   [9]  add [L0] [8]
   [10] load @([7]+[9])
 3 [11] const 1
 4 [12] add [10] [11]
 7 [13] load @([L0]+8)        ❶
   [14] cmp [L2] [11]
   [15] cbranch_ult [14]
   □        --> [17]
   [16] call bad_array_idx
   [17] target
   [18] const 2               ❶
   [19] sll [L2] [18]
   [20] const 12
   [21] add [L0] [20]
   [22] store @([19]+[21])    ❶
```

Optimized intermediate representation

```
bpc eid
 2 [1]  load @([L0]+8)
   [2]  cmp [L1] [1]
   [3]  cbranch_ult [2]
 □          --> [5]
   [4]  call bad_array_idx
   [5]  target
   [6]  const 2
   [7]  sll [L1] [6]
   [8]  const 12
   [9]  add [L0] [8]
   [10] load @([7]+[9])
 3 [11] const 1
 4 [12] add [10] [11]
 7 [14] cmp [L2] [11]
   [15] cbranch_ult [14]
 □          --> [17]
   [16] call bad_array_idx
   [17] target
   [19] sll [L2] [18]
   [22] store @([19]+[9])
```

Figure 4. Array accesses bytecodes are decomposed so that optimizations can be performed on their components.

3.3.4 Global Optimization

We perform global optimizations non-iteratively, but produce results that are equivalent to a traditional iterative dataflow analysis. This is possible because IR expression generation for EBBs is processed in reverse post-order traversal (an EBB must be processed before any of it sucessors have been). In this fashion, we can propagate forward flow information to successor EBBs before IR expressions are generated for them.

At EBB headers, we merge flow information for global copy and constant propagation before generating IR expressions in the EBB. Loop invariant code motion and global CSE are handled within the `EXPR_add()` function introduced in the previous section.

If the current BB exists within a loop, a check is made to see if the new expression can be hoisted to the loop preheader. For most expressions, this involves determining if their arguments are constants or locals that are not redefined in the loop. This later property is queried from the loop locals and field access statistics computed during the CFG generation pass (Section 3.2).

Global CSE is performed on a new expression only if local CSE fails and its arguments are locals or globals that we have created. The global CSE optimizer searches BBs backwards toward the method entry for matching available expressions using the same routines used for CSE within an EBB. We terminate the search if one of the arguments is redefined along any of the backward paths toward method entry.

We compare our local and global optimizer to other implementations in Section 4.1.

3.3.5 Data Flow Statistics

During DFG generation, we collect additional information that will be utilized by the register allocator. Each IR expression includes a local and global use counter. The local use counter is incremented whenever a given expression is used as a source in another expression. We also compute a flag called `expr_spans_call` which is set if a call occurs between an expression definition and use.

The global use counter is accumulated towards the expression definition after all expressions have been generated using a post-order traversal from method exit BBs to the entry BB. The global use count for a given local expression at the exit of a BB is equal to the sum of the local and global uses for the local expression at the entry to immediate successor BBs. We do not consider this scan a major pass as only expressions in Java locals at BB exits and entries are considered. This computation is equivalent to a live variable dataflow analysis but also includes relative weighting of uses. This computation must be iterated when loops are present to compute correct liveness values within the loop.

Also computed concurrent with DFG generation is a structure we call a BB join. A BB join is the union of all adjoining BBs entries or exits, with each BB identified as whether its entry or exit is part of the join. Example BB joins are shown in Figure 6. A BB may be in at most two BB joins, or it may be in only one BB join (its entry and exit share common successors and predecessors). A BB join is used to link register assignments between dependent BB entry and exit points, described in more detail in Section 3.4.2.

3.4 Code Generation Pass

3.4.1 Code Generation

For the most part, code is generated in place using single pass of the expressions in a BB. Like expression generation, EBBs are processed in reverse post-order when generating code. A patching system is used to fix unknown values likes branch targets and variable sized method prologues and epilogues (for certain ISAs) after the primary code generation pass. Selective code buffering and movement is supported for block-level code scheduling. This facility is currently used to move array out-of-bounds throw clauses out of the critical code path and to implement loop inversion.

3.4.2 Register Allocation

We do not allocate registers in a separate pass, but assign registers as code is generated. Use counters, the `expr_spans_call` flag, and register pre-assignments are all critical to achieving good register allocation. Pseudo-code for the register allocator is shown in Figure 5 and an example allocation pass is shown in Figure 6.

```
REG_alloc( expr * e )
{
    reg * r; int r_type;

    // HANDLE REGISTER RESERVATIONS
    if( e->reserved_reg ){
        r = e->reg;
        if( r->is_assigned ){
            r->spill_expr();
            r->free();
        }
        r->assign( e );
    }
    // NORMAL ALLOCATION
    else{
        // CLASSIFY REG ASSIGNMENT
        if( e->uses.other > 0
            || e->spans_next_call ){
            r_type = R_caller_saved;
        }
        else{
            r_type = R_temp;
        }
        // ASSIGN REGISTER
        if( r = get_free_reg( r_type ) ){
            r->assign( e );
        }
        else if( r = any_free_reg() ){
            r->assign( e );
        }
        else if( r = min_cost_live_reg() ){
            r->spill_expr();
            r->free();
            r->assign( e );
        }
    }
}
```

Figure 5. `REG_alloc()` pseudo code (simplified code).

Register allocation starts with any register allocated arguments at a method entry. When a register is needed for an expression being generated, allocation occurs as follows. If a register has not been pre-assigned, we must choose an appropriate register class assignment (e.g. temporary or call-preserved register). Accounting for whether an expression must survive a future call (`expr_spans_call` flag), whether it will survive past the BB it is defined in (global use counter), and potential conflicts with registers allocated at the BB exit, we can select an appropriate register to minimize future moves or spills.

As each expression is processed and code is generated for it, we decrement the local use counter of the source arguments to reflect that the argument has been "used." When the local use counter and the global user counter both are zero for a register allocated expression, the register can be freed as it can be guaranteed that this expression will never be used again.

Once code has been generated for a BB, the BB linker is responsible for properly linking register assignments between BBs. The register assignments at a BB exit are compared to the assignments in the BB join. Register assignments are added to the BB join or a register move or spill is generated match a previous BB join register assignment.

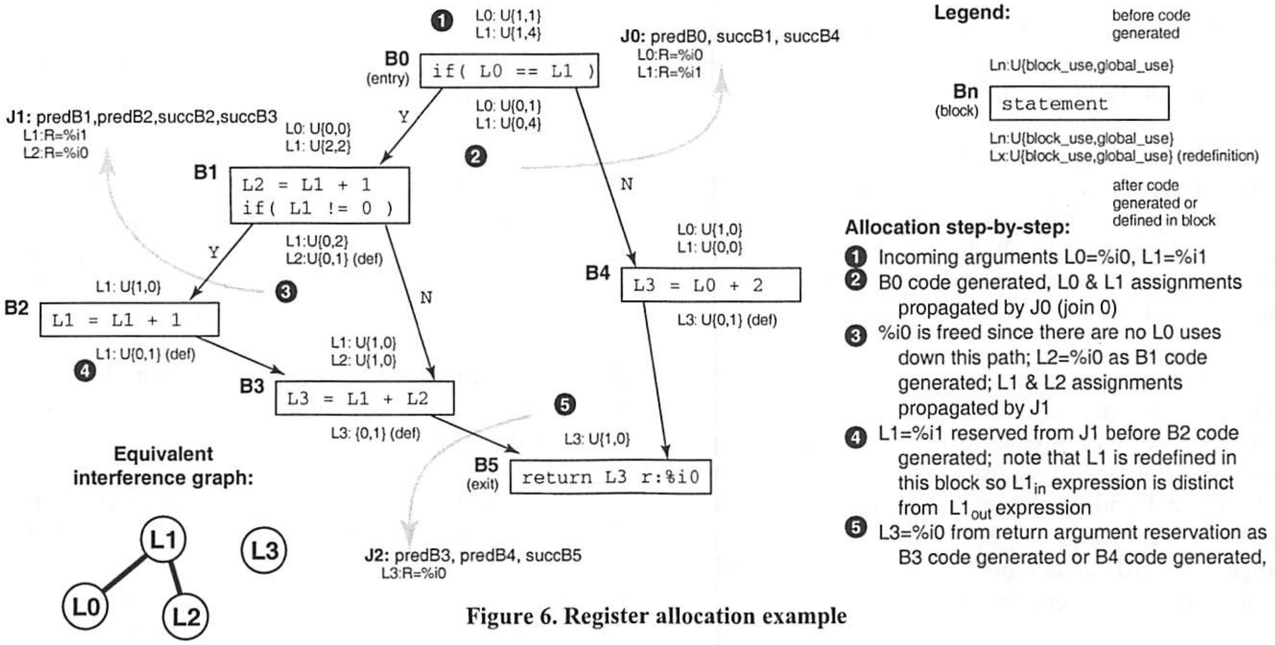
We will compare our register allocator to other schemes in Section 4.2

3.4.3 Instruction Scheduler

We use a standard list scheduler for low-level instruction scheduling. To simplify the implementation, scheduling is currently limited to a given BB. Despite this limitation, we believe BB regions have enough instructions to sufficiently target the biggest benefactors of scheduling, filling load and branch delay slots. We schedule instructions only after they have been generated because some IR expressions may expand into more than one instruction while others may not even generate one. Additionally, the loads and spills to the runtime frame of Java locals, which are only generated as needed, are not represented as explicit expressions in our IR.

3.5 Fast Optimization of Machine Idioms

Machine idioms are instructions or instruction sequences for a specific ISA that execute more efficiently than a similar sequence of instructions targeted for a more general architecture. Common machine idioms include immediate arguments, auto-increment arguments, call argument specifics, leaf



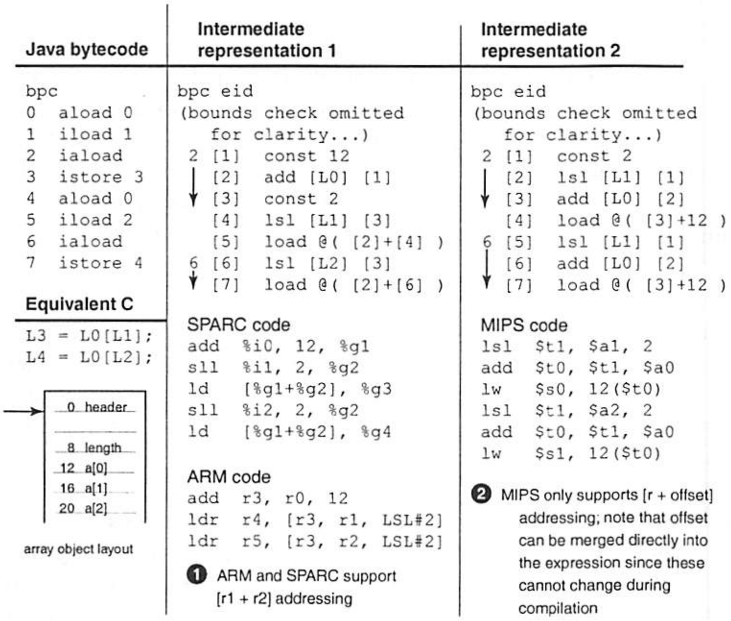
Allocation step-by-step:

1. Incoming arguments L0=%i0, L1=%i1
2. B0 code generated, L0 & L1 assignments propagated by J0 (join 0)
3. %i0 is freed since there are no L0 uses down this path; L2=%i0 as B1 code generated; L1 & L2 assignments propagated by J1
4. L1=%i1 reserved from J1 before B2 code generated; note that L1 is redefined in this block so $L1_{in}$ expression is distinct from $L1_{out}$ expression
5. L3=%i0 from return argument reservation as B3 code generated or B4 code generated,

Figure 6. Register allocation example

procedure optimization, and condition code (CC) usage.

Optimizing for machine idioms is often left to the end of the compilation process, after one has generated machine specific code, using a peephole optimizer [12]. A peephole optimizer searches code for instruction patterns it knows how to replace with an equivalent sequence that requires less instructions or cycles. Using peephole optimization is expensive because it usually requires at least one additional pass across all instructions generated with a pattern matcher.



```
bpc
0  aload 0
1  iload 1
2  iaload
3  istore 3
4  aload 0
5  iload 2
6  iaload
7  istore 4
```

```
L3 = L0[L1];
L4 = L0[L2];
```

```
bpc eid
(bounds check omitted
   for clarity...)
 2 [1]  const 12
   [2]  add [L0] [1]
   [3]  const 2
   [4]  lsl [L1] [3]
   [5]  load @( [2]+[4] )
 6 [6]  lsl [L2] [3]
   [7]  load @( [2]+[6] )
```

SPARC code

```
add  %i0, 12, %g1
sll  %i1, 2, %g2
ld   [%g1+%g2], %g3
sll  %i2, 2, %g2
ld   [%g1+%g2], %g4
```

ARM code

```
add  r3, r0, 12
ldr  r4, [r3, r1, LSL#2]
ldr  r5, [r3, r2, LSL#2]
```

1. ARM and SPARC support [r1 + r2] addressing

```
bpc eid
(bounds check omitted
   for clarity...)
 2 [1]  const 2
   [2]  lsl [L1] [1]
   [3]  add [L0] [2]
   [4]  load @( [3]+12 )
 6 [5]  lsl [L1] [1]
   [6]  add [L0] [2]
   [7]  load @( [3]+12 )
```

MIPS code

```
lsl  $t1, $a1, 2
add  $t0, $t1, $a0
lw   $s0, 12($t0)
lsl  $t1, $a2, 2
add  $t0, $t1, $a0
lw   $s1, 12($t0)
```

2. MIPS only supports [r + offset] addressing; note that offset can be merged directly into the expression since these cannot change during compilation

Figure 7. Idiomatic optimizations can be performed more quickly by having different IRs for different ISAs.

In our dynamic compiler, we handle machine idioms as well as other miscellaneous opportunities to reduce instruction sequences by providing machine dependent (MD) code an opportunity to access an expression as soon as it is created. This allows us to perform preliminary analysis and set flags which can be accessed at code generation time to help generate machine idioms. A common use of this facility is for generating immediate arguments. In the `md_prep` access, the MD code can flag constants that can fit into immediate fields for that ISA. When the actual pass to generate code occurs, only constants that cannot fit into immediate fields will be register allocated.

Another important use of this facility is for satisfying calling argument conventions. We implement a register reservation system where register assignments can be made before the code generation pass. Using this system, we can pre-assign registers to expressions that will be used as a call argument, which prevents a possible expression assignment to an unknown register and then an extra move to the correct argument register during code generation.

For more complex idioms, we can make minor adjustments to the IR to simplify certain optimizations. For example, we provide a special branch IR representation to accommodate architectures that set CCs. Also useful is altering array access decompositions to target addressing modes available for the ISA, as illustrated in Figure 7.

Using this approach, we do not need to make an additional pass to optimize for machine idioms. This system appears sufficiently robust, as we have been able to accommodate all the low-level optimizations that we have wanted to perform on the ISAs we have targeted so far.

Initial design of the compiler was done on a MIPS IV ISA. So far, we have ported the compiler to the SPARC v9 and StrongARM ISAs. An ISA port requires defining a register file, assigning register classes and coding `md_prep` (when required) and `md_asm` functions for each IR expression type (see Figure 1). MD code represents about 1/4 – 1/3 of the total binary size of our dynamic compiler.

As a demonstration of the portability of our design, each port, with some MD optimizations, only took about 2 man-weeks to complete. As RISC style machines, the SPARC and MIPS ports are similar, but there are still significant differences in the register file and calling convention models. The ARM architecture provides for unusual source argument arrangements, load/store addressing modes, and full instruction predication. The current port does not use the more exotic aspects of the ARM architecture, but support could be added in future revisions. A x86 port is planned, though not currently implemented.

4. Comparisons To Other Compilers

4.1 Dataflow Analysis

Our compiler implementation of dataflow algorithms differs significantly from most modern optimizing compilers. In this section, we discuss how our approach compares against traditional implementations.

Optimizing compilers like the Sun-server compiler [6] and LaTTe JIT compiler [19] implement traditional dataflow analysis using lattices and flow functions [12]. Traditional implementations can be computationally expensive for several reasons. Setting up a problem for dataflow analysis often requires scanning the entire method to set up initial conditions. Additionally, some optimizations require more involved auxiliary data structures than bit vectors alone, may be iterative, or may require solving more than one dataflow problem.

Rather than setting up separate dataflow problems, we apply several optimizations concurrently as IR expressions are generated. Forward flow information required by these optimizations is communicated by processing EBB in reverse post-order when generating IR expressions

When performing traditional iterative dataflow analysis, blocks are also processed in reverse post-order to minimize required iterations to reach a fixed point [12]. If A is the maximal number of loop back edges in the CFG, the bound on the maximum number of times a block may be visited before reaching a fixed point is $A+1$. Logically, this is required to propagate forward flow information through loops so a fixed point can be found at the loop header. In our implementation, we can use the per loop Java local load and store usage statistics collected in the CFG generation pass (Section 3.2) to compute forward flow information through loop back edges without iterating. For example, a local V can be copy or constant propagated to successor blocks outside the loop if V is not redefined within the loop (no stores to V within the loop).

Our loop invariant code motion optimizer is also non-iterative. Normally, a loop has to be scanned multiple times as loop invariant code motion of one expression inevitably can cause other expressions dependent on it to become loop invariant [12]. We generate IR expressions in a BB in order, and predecessor loop BBs are always processed first. As a result, when loop invariant code motion is applied to an expression, this change is immediately communicated to successive, dependent instructions of the loop.

What might be considered a major limitation to our approach is that it is largely restricted to optimizations that rely primarily on forward flow information. Since most basic optimizations are based on forward flow information (e.g. reaching definitions, available expressions, copy propagation, constant propagation) [12], we do not consider this a serious restriction.

4.2 Register Allocation

In this section, we compare our on-the-fly register allocation scheme with graph coloring and other proposed fast allocation schemes. Most register allocation algorithms work with liveness information, the span between a variable's definitions to all its uses.

While graph coloring usually generates the best results, it can be expensive. Register coalescing and spill points cause the algorithm to iterate, which can result in high register allocation times, particularly for methods that are large or are not initially colorable [12].

Table 2. Features and characteristics of compilers evaluated.

JIT	Sun - Client	Sun - Server	SNU LaTTe	MicroJIT
Source	C++	C++	C	C
64b ops	Full	Full	Some	Some
Intermediate representation	Simple	SSA dataflow	Dataflow	Dataflow
Major compiler passes	4	Iterative	7	4
Optimizations	Block merging/elimination Simple constant propagation Inlining & specialization	Loop invariant code motion Global value numbering Conditional constant propagation Inlining & specialization Instruction scheduling	EBB value numbering EBB constant propagation Loop invariant code motion Dead code elimization Inlining & specialization Instruction scheduling	CSE Copy propagation Constant propagation Loop invariant code motion Dead code elimization Inlining & specialization Instruction scheduling
Register allocation	1-pass dynamic	Graph coloring	2-pass dynamic	1-pass dynamic
Garbage collection	Incremental copying	Incremental copying	Incremental mark & sweep	Incremental mark & sweep
Compiler size	700KB	1.5Mb	325KB	200KB
Interpreter size	220KB	220KB	65KB	None

All the fast register allocation schemes share more with each other than with graph coloring. The most important common characteristic is that they consider allocation using a limited view of interference. Additionally, these algorithms handle spills dynamically at points in the program where there are no free registers.

LaTTe's register allocation system probably bears the closest resemblance to our implementation [19]. Their algorithm uses three passes for each block: a scan that computes estimates of live variables and their last uses, a backward sweep that computes preferred register assignments, and a final forward sweep which allocates registers and removed unnecessary copies.

Compared to the LaTTe JIT, our on-the-fly allocator requires one less pass over the code. We compute liveness (derived from our local and global use counters) and perform register preallocation in the same step. Additionally, we integrate these two computations into the DFG generation stage so that only one pass is required for register allocation (concurrent with code generation). At each allocation point, the LaTTe allocator also has less information to make good spill decisions and register class selections.

Linear scan allocators direct global allocation of register using a linear sweep of the program being compiled [4][17]. The basic linear scan allocator uses a simple view of liveness known as a lifetime interval, which spans from a variable definition to where it is last live in linear program order. Each step of the algorithm tracks the active lifetimes at a given program point. When there are more active lifetimes than available registers, the longest active lifetime is spilled.

Although the basic linear scan algorithm is probably the fastest allocator we describe here, its representation of liveness is probably the most imprecise. An algorithm has been proposed to improve the precision of lifetime intervals [17], at the cost of an additional dataflow analysis pass, and a scan to resolve assignment conflicts at basic block boundaries.

5. Experiment Results

5.1 Setup

The microJIT was developed on a commercial version of the open source Kaffe virtual machine (http://www.transvirtual.com). We compared our JIT compiler against three other compilers that target the SPARC ISA. Characteristics of these compilers are shown in Table 2. The SPARC architecture was chosen because it had the largest availability of good compilers for which source code could be found, and because we wanted performance results from a neutral RISC architecture. We could have also chosen the x86 platform, but we were concerned that its small register file might skew the results by eliminating most possibilities for register allocation.

We included the two dynamic compilers from the Sun JDK, the client and server compilers [9][6]. The server compiler uses the powerful, but expensive static-single assignment (SSA) representation internally. This compiler is not optimized for fast compilation times, but generates extremely good code through traditional dataflow analysis. The client compiler does not perform any advanced analysis, but focuses on basic register allocation and inlining optimizations. Both of these compilers run under the HotSpot VM, which only compiles frequently called methods and interprets otherwise.

The other compiler included in our experiment is the LatteVM [19]. This relatively fast dataflow compiler implements many of the optimizations performed by the Sun-server compiler. This VM also supports lazy compilation, although it appears to use very little interpretation during exeuction. This compiler was not designed to be ported to different ISAs as the IR maps closely to the SPARC ISA.

We used perfmon (http://web.cps.msu.ed/~enbody/perfmon.html), a library interface to the UltraSparc2 hardware counters, to time compilations. This was necessary to get accurate times for the compilation of smaller methods. The UNIX time command was accurate enough for code execution times. All VMs ran on the same machine (200MHz UltraSparc2 w/ Solaris 8).

Table 3 lists the benchmarks used to evaluate the performance. In choosing benchmarks, we tried to include a variety of programs to represent both numerical and object-oriented programs. Most the larger benchmarks are part of the Spec JVM98 (http://www.spec.org/) and Java Grande (http://www.epcc.ed.ac.uk/javagrande/javag.html) benchmark suites. Scimark2 and jBYTEmark are benchmarks suites comprised of smaller kernels.

Table 3. Method bytecode sizes by benchmark.

Benchmark	Method bytecode size				
	<50B	50-250B	250B - 1KB	1K – 5KB	>5KB
mp3 – mp3 decoder	131	70	17	4	1
mtrt – raytracer	128	34	13	2	1
jess – expert system	289	60	11	0	0
compress – compression	35	8	4	0	0
db – database	16	9	0	0	0
jlex – parser gen.	47	33	23	3	3
deltablue – planner	52	15	1	0	0
richards – task simulator	306	54	4	0	0
java_cup – parser gen.	122	30	7	0	0
moldyn – particle simulation	13	15	3	1	0
search – alpha beta search	15	20	4	0	0
h263dec – video decoder	40	24	20	11	3
pizza – java compiler	327	194	40	9	0
euler – fluid dynamics	14	14	4	5	0
jpeg – image compression	237	75	54	16	0
mips_sim – cpu simulator	25	25	9	2	0
scimark2 – fp loops	14	13	2	0	0
jbytemark – int & fp loops	47	64	15	0	0

5.2 Compilation Time

Compilation times, decomposed by method bytecode size, are shown in Figure 8. We normalized to bytecodes processed per 1k cycles to accommodate varying method bytecode sizes within each bin. In this figure (and subsequent figures), two bars show microJIT's compilation performance: advanced optimizations (scheduling, inlining, loop opt) are enabled for microJIT1 and disabled for microJIT2. For methods <1000b, microJIT's compilation times are always better, averaging over 2.5x faster than the closest dataflow compiler, LaTTe, and 12x faster than the Sun-server compiler. Relative compilation times may be even better against the LaTTe compiler because it is heavily optimized for SPARC, and may incur additional overheads to support multiple target ISAs. While we are only about 30% faster than the Sun-client compiler, the client compiler performs fewer optimizations than the microJIT, LaTTe, and Sun-server compiler.

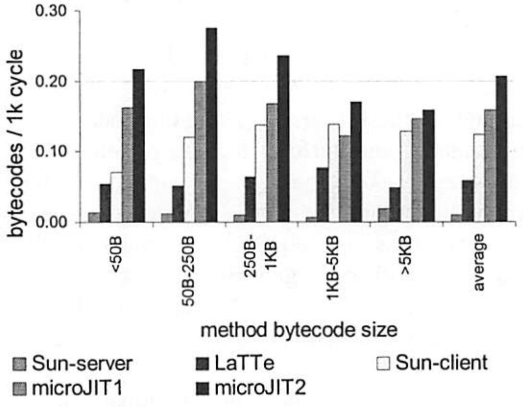


Figure 8. Compilation rate for given method bytecode sizes in bytecodes processed per 1k cycles.

Figure 9 shows the effect of various optimizations on compile time (CSE is always on by default). Inlining results in about a 10% slowdown and scheduling caused an average 15% penalty. The cost of loop optimizations are relatively cheap expect for large methods, where it adds over 20% to the compilation time.

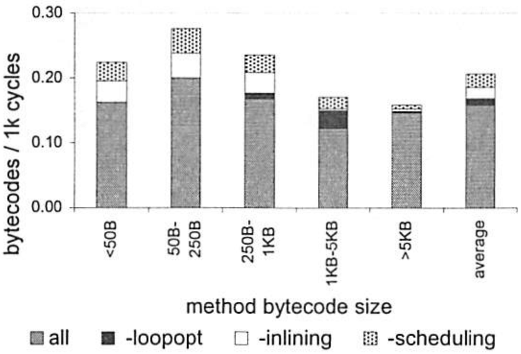


Figure 9 Compilation rate for microJIT with different optimizations disabled, in bytecodes processed per 1k cycles.

Figure 10, also decomposed by method bytecode size, breaks down time spent in each pass of our compiler. For very large methods (>1000b), time spent in DFG generation dominates almost 70% of compilation time. We attribute this shift to the high cost of CSE. As methods get large, we expect regions searched for CSE will grow, along with the number of expressions on which CSE is applied, resulting in non-linear computational cost. We believe this is also the primary cause of decreasing compilation speeds for larger methods. If this effect is undesirable, one possible fix is to limit the depth of backward searches performed by CSE.

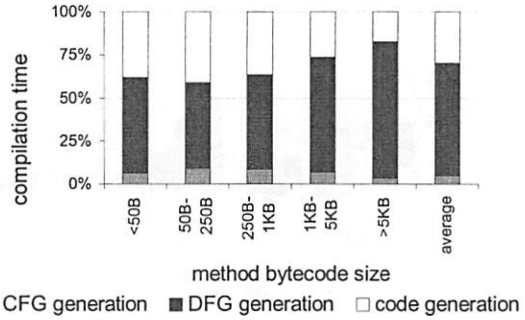


Figure 10. Times spent in each pass of microJIT.

5.3 Generated Code Performance

Performance of the code generated by the compilers is shown in Figure 11 (long running applications, large data sets) and Figure 12 (short running applications, small data sets). Benchmarks included in both graphs (like compress, db, jess, mp3, and mtrt) are run with different input data set sizes. Reported performance times are for total running time, including compilation, interpretation (if any), garbage collection, and native execution. For comparison, performance of the original Kaffe JIT and a Sun JDK using only interpretation are also included. The performance in Figure 11 is expressed as speedup normalized to the JIT compiler used in the JDK1.1 to compensate for the different sizes of each benchmark. This JIT compiler is a suitable baseline because it is a relatively conservative dynamic compiler that does not attempt any advanced optimizations. Short run execution times in Figure 12 are represented in seconds and have been divided into interpretation, compilation

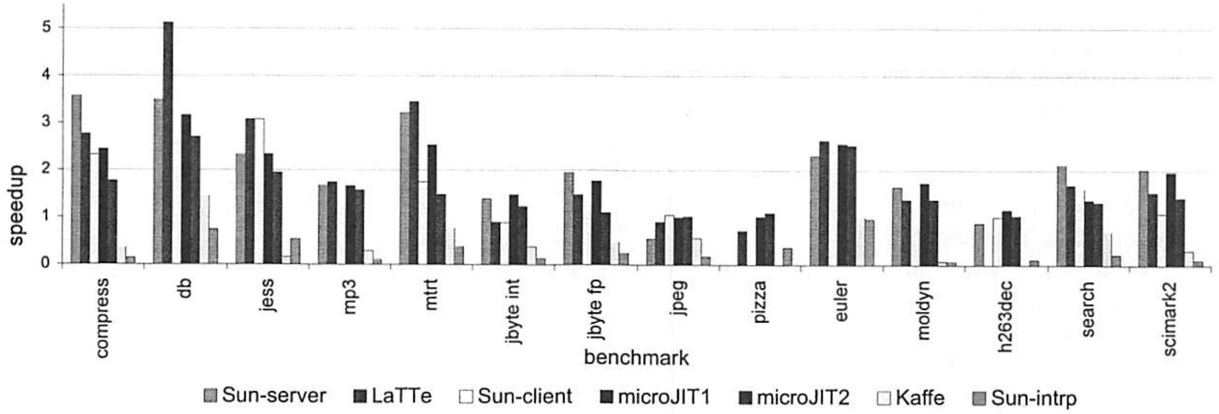


Figure 11. Speedup of large benchmarks relative to JDK1.1 (Sun's pre-Hotpot JIT).

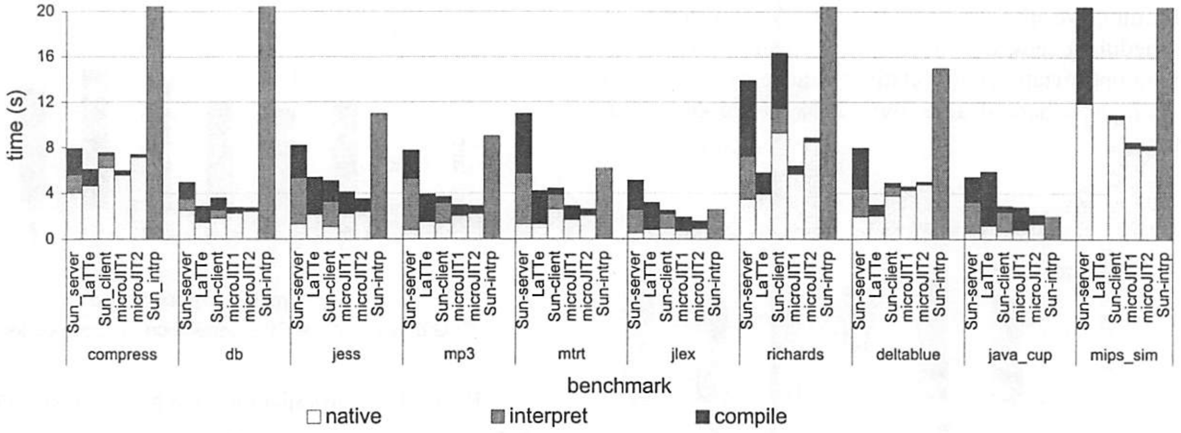


Figure 12. Performance on short running benchmarks.

and native execution times. (Note: missing bars in the graphs represent programs that we could not collect results for)

For long running applications, microJIT performs well on numerically intensive applications (e.g. mp3, euler, moldyn, jpeg, h263dec, jBYTEmark, and scimark2). While the LaTTe and Sun-server compilers still produce better code, microJIT is able to outperform the Sun-client compiler on many applications. On short running applications, microJIT's low compilation times allow it to keep total execution time small relative to the other systems.

Overall, we are most disappointed by our performance on object-oriented applications like db and jess. db's execution time is largely dominated by a loop nest within a shell sort routinue. For this benchmark, we suspect aggressive array bounds check elimination within the loop nest allows LaTTe to perform particularly well on this program.

To understand further the quality of code generated by our compiler, we also decomposed execution times of some of the long running benchmarks, given in Figure 13. Results are normalized to the Sun-server compiler and include garbage collection times. This graph suggests that one factor limiting performance of microJIT code is inefficiencies in our garbage collector. On applications that allocate memory intensively, our system spends a larger percentage of time in collection than other virtual machines, deteriorating its relative performance.

Another performance limitation could be from our naïve implementation of specialization. The Sun-client and server compilers support a particularly fast form of specialization using class hierarchy analysis (CHA) and deoptimization [9]. Non-final, public virtual and interface calls that have only one target class can be inlined directly without a check to verify the correct target object class. If dynamic class loading causes this virtual or interface call to have more than one target class, the methods can be recompiled with these optimizations removed, including those on the current

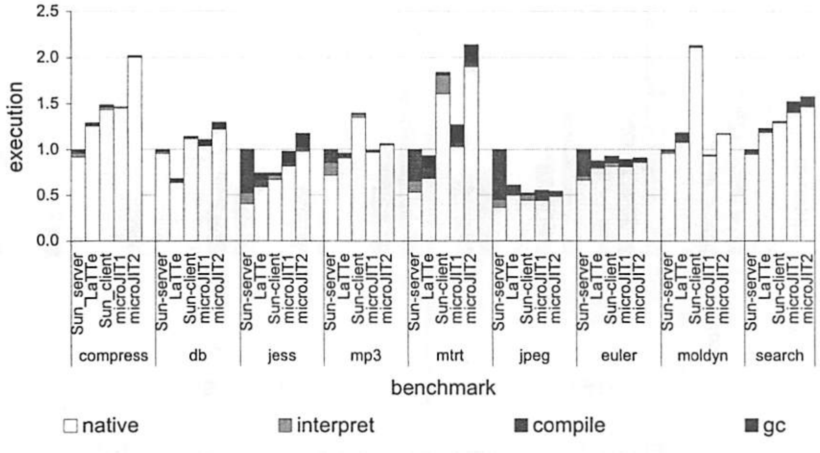


Figure 13. Performance of large benchmarks normalized to Sun-server.

call stack, so that the program will run correctly.

5.4 Static Memory Usage

Total size of the compilers and associated interpreter (if any) are shown in Table 2. These numbers were obtained by taking associated object files and applying the UNIX `strip` to them to remove unnecessary symbols. At 200KB, microJIT's static memory requirements are smaller than the other compilers. Total static memory requirements are further improved by omission of an interpreter in our system. While Sun-server and Sun-client may be larger because they are written in C++, and support the profiling (JVMPI) and debugging (JVMDI) interfaces, we believe these differences should not dramatically affect static memory comparisons.

5.5 Dynamic Memory Usage

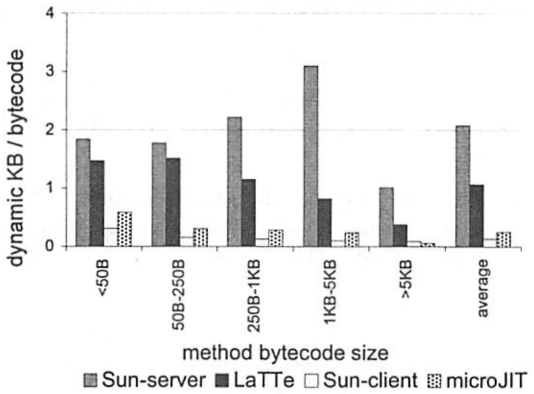


Figure 14. Dynamic memory required during compilation.

An important consideration for dynamic compilation in an embedded system is the limited dynamic memory available to the compiler. Figure 14 shows dynamic memory used by the compilers during compilation. On average, microJIT uses 25% of the memory required by the LaTTE compiler and 12.5% of the memory required by the Sun-server compiler, but it uses twice the memory required by the Sun-client compiler. These numbers suggest a 250KB buffer is sufficient memory for microJIT to compile method bytecodes less than 1KB.

To limit the dynamic memory required by the compiler for larger method bytecodes (> 1KB), microJIT could be amended to support partial compilation. In this mode, microJIT's first pass (CFG construction) would execute normally and generate the CFG of all the BBs in the method. The DFG generation and code generation passes would then execute as before, but only on sections of the CFG at one time (e.g. one EBB or loop nest). This relies on the observation that the bulk of dynamic memory used by the compiler is for the intermediate representation of the bytecodes. By only generating the intermediate representation for subsections of a method at one time, we can reduce total dynamic memory requirements. This possible improvement to microJIT would reduce dynamic memory requirements at the cost of limiting some global optimizations for large bytecode methods.

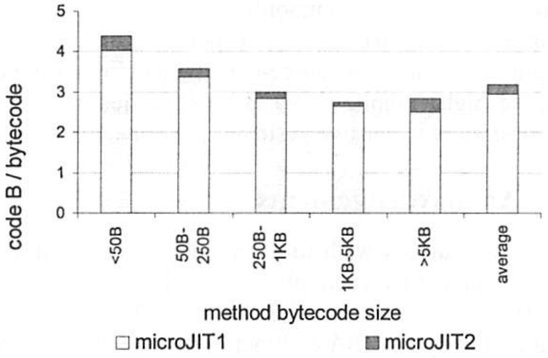


Figure 15. Code expansion of native code after translation of bytecodes.

The other important dynamic memory consideration in an embedded system is memory used to store translated code. The effects of a limited code buffer on total system performance were beyond the scope of our study (e.g. choosing which translated methods to discard and factoring the cost of recompilation when the code buffer is full), but we did collect statistics on code expansion resulting from translation. Figure 15 shows the average number of bytes of native code (code and data segments) generated per bytecode translated by mciroJIT. microJIT1 (with all optimizations enabled) generated less code primarily due to filling of branch delay slots by the instruction scheduler. We also did not observe any dramatic differences in code expansion between the compilers evaluated. We found the largest benchmarks evaluated here (jpeg and pizza compiler) generated at most 300KB of native code.

6. Conclusions

We have demonstrated how a fast dynamic optimizer can be constructed that includes advanced optimizations without incurring high compilation costs or having high memory requirements. This was accomplished by minimizing compiler passes while optimizing aggressively and by efficient communication and representation of flow information. Unlike traditional dataflow compilers that solve dataflow equations and apply optimizations successively, we perform local and global optimizations as the IR expressions are generated. Additionally, we allocate registers concurrently with code gneration using an on-the-fly allocator that utilizes local interference, liveness, and register classes when making allocation and spill decisions.

Our experiment shows that the tradeoff between short compile times and high code quality may be less pronounced than commonly believed. This result suggests that we can incorporate small dynamic compilers into resource-constrained environments where high compile times, poor code quality, and the cost of more expensive systems cannot be tolerated.

7. Acknowledgements

The authors wish to acknowledge Tim Wilkinson and Transvirtual Technologies for their support of the Kaffe JVM used in this study. This work was supported by DARPA contract MDA904-98-C-A933.

8. References

[1] Adl-Tabatabai, A.R. et al. Fast, Effective Code Generation in a Just-In-Time Java Compiler. In SIGPLAN'98, Montreal, Canada, 1998.

[2] Alpern, B. et al. Implementing Jalapeño in Java. In OOPSLA'99, Denver, Colorado, November, 1999.

[3] Bala, V., Duesterwald, E., and Banerjia, S. Dynamo: A Transparent Dynamic Optimization System. In PLDI'00, Vancouver, BC, Canada, June, 2000.

[4] Blickstein, D.S. et al. The Gem Optimizing Compiler System. Digital Equipment Corportation Technical Journal, 4(4):121-135, 1992.

[5] Cierniak, M., Lueh, G. Y., and Stichnoth, J. Practicing JUDO: Java Under Dynamic Optimizations. In PLDI'00, Vancouver, BC, Canada, June, 2000.

[6] Click, C. High-Performance Computing with the Server Compiler for the Java HotSpot Virtual Machine. In JavaOne 2001, San Francisco, CA, June, 2001.

[7] Engler, D.R. vcode: a retargetable, extensible, very fast dynamic code generation system. In PLDI'96, Philadelphia, PA, May, 1996.

[8] Gosling, J., Joy, B., and Steele, G. The Java Language Specification. Addison Wesley, Reading, MA, 1996.

[9] Griessemer, R. and Mitrovic, S. The Java HotSpot Virtual Machine Client Compiler: Technology and Application. In JavaOne 2001, San Francisco, CA, June, 2001.

[10] Holzle, U. et al. Java On Steroids: Sun's High-Performance Java Implementation. In Hot Chips '97, Stanford, CA 1997.

[11] Lindholm, T. and Yellin, F. The Java Virtual Machine Specification. Addison Wesley, Reading, MA, 1997.

[12] Muchnick, S. Advanced Compiler Design Implementation. Morgan Kaufmann Publishers, San Francisco, CA, 1997.

[13] Poletto, M. Language and Compiler Support for Dynamic Code Generation. PhD thesis, MIT, 1999.

[14] Poletto, M., Engler, D.R., and Kaashoek, M.F. tcc: A System for Fast, Flexible, and High-level Dynamic Code Generation. In PLDI'97, Las Vegas, NV, June, 1997.

[15] Rubin, N. and Chernoff, A. Digital FX!32: A Utiliity for Fast Transparent Execution of Win32 x86 Applications on Alpha NT. In Hot Chips '97, Stanford, CA, Auguest, 1997.

[16] Suganuma, T. et al. Overview of the IBM Java Just-in-Time Compiler. In IBM Systems Journal, Vol. 39, No. 1, 2000.

[17] Traub, O., Holloway, G., and Smith, M.D. Quality and Speed in Linear-scan Register Allocation. In SIGPLAN'98, Montreal, Canada, 1998.

[18] Witchel, E. and Rosenblum, M. Embra: Fast and Flexible Machine Simulation. In ACM SIGMETRICS '96, Philadelphia, PA, 1996.

[19] Yang, B.S. et al. LaTTe: A Java VM Just-in-Time Compiler with Fast and Efficient Register Allocation. In PACT'99, New Port Beach, CA, October, 1999.

Supporting Binary Compatibility with Static Compilation[*]

Dachuan Yu Zhong Shao Valery Trifonov
Department of Computer Science, Yale University
New Haven, CT 06520-8285, U.S.A.
{yu, shao, trifonov}@cs.yale.edu

Abstract

There is an ongoing debate in the Java community on whether statically compiled implementations can meet the Java specification on dynamic features such as binary compatibility. Static compilation is sometimes desirable because it provides better code optimization, smaller memory footprint, more robustness, and better intellectual property protection. Unfortunately, none of the existing static Java compilers support binary compatibility, because it incurs unacceptable performance overhead. In this paper, we propose a simple yet effective solution which handles all of the binary-compatibility cases specified by the Java Language Specification. Our experimental results using an implementation in the GNU Java compiler shows that the performance penalty is on average less than 2%. Besides solving the problem for static compilers, it is also possible to use this technique in JIT compilers to achieve an optimal balance point between static and dynamic compilation.

1 Introduction

Modern software applications are often built up by combining many components. Some of these components are shared libraries which allow multiple applications to share large amounts of system software.

Shared libraries evolve over time so that new functionality can be added, bugs can be fixed, algorithms and efficiency can be improved, and deprecated functions can be removed. Evolving or modifying these libraries can affect applications that depend on them, thus library evolution may cause compatibility problems.

However, it is usually undesirable to recompile a whole application just to accommodate the changes in a single component. In the case of widely distributed libraries, used by many unknown applications, it is often impractical or impossible to recompile even only the importing units. A popular current approach is to try to guarantee that binaries can be directly replaced by compatible binaries without compromising a working system.

Binary compatibility is a concept introduced to address this problem. It was initially referred to as *release-to-release binary compatibility* [11], and later defined in the Java Language Specification (JLS) [12], which describes the changes that developers are permitted to make to a package or to a class or interface type while preserving compatibility with existing binaries. Thus the Java binary compatibility prescribes conditions under which modification and recompilation of classes do not necessitate recompilation of other classes depending on them.

In the Java Virtual Machine [20], support for binary compatibility is primarily due to the use of symbolic references to look up fields and methods at run-time. However, in some cases a native compiler for Java is needed that compiles Java (or bytecode) programs directly into native code in the same manner as compilers for C/C++. This ahead-of-time compilation is desirable because it yields better optimized code, more robust deployed applications, and offers better intellectual property protection [3, 5, 7]. We will elaborate on this later.

Nevertheless, supporting binary compatibility with ahead-of-time compilation is a hard problem because of the seemingly contradictory requirements. When certain changes are allowed due to binary compatibility, the contents of a class cannot be completely determined until the class is loaded. However, ahead-of-time compilers usually generate hard-coded offsets based on the layout information of other classes at compile time.

A well-known problem is that the standard compilation techniques for virtual methods in object-oriented languages preclude binary compatibility (cf. the fragile base class problem [13, 26]). For example, the documen-

[*] This work is supported in part by DARPA OASIS grant F30602-99-1-0519, NSF grant CCR-9901011, and NSF ITR grant CCR-0081590. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

tation on binary compatibility [30] in the EPOC C++ System says:

> ... virtual member functions are for life—you can't add or remove virtual member functions, or change the virtuality of a function, or change their declaration order, or even override an existing function that was previously inherited, ...

For compliance with the binary-compatibility requirements of Java some existing native compilers solve this problem by generating (at least) some of the code at run time, which unavoidably negates some of the benefits of pre-compilation. Other existing native compilers simply have no support for binary compatibility, because the obvious solutions (e.g. method lookup by name at run time) seem to incur high performance overhead.

This paper presents a simple yet effective solution using static compilation, which meets all Java binary compatibility requirements with little performance penalty. The contributions are:

- In our solution, the compilation is fully static, which allows the compiler to take advantage of the well-developed static compilation techniques for better code optimization.
- Our solution covers all the cases specified in the JLS. Different features—including methods, fields, interfaces, and modifiers—are supported by the same set of simple core techniques.
- Our solution also detects all binary-incompatible changes and gracefully raise proper exceptions at load or run time.
- Our solution is efficient. We describe an implementation in the GNU Java compiler (GCJ). The performance test shows that the performance penalty of our new technique is on average less than 2%.

In the remainder of this introduction, we briefly describe the benefits of static compilation.

1.1 Why Static Compilation?

Two popular approaches for compiling Java programs are Just-In-Time (JIT) compilation (e.g. Sun Hotspot [29], Cacao [18], OpenJIT [24], shuJIT [28], vanilla Jalapeno [1]) and static compilation (e.g. BulletTrain [22], Excelsior JET [10], GCJ [32], IBM VisualAge for Java [14], JOVE [15]). It would be wrong to say one approach is definitely better than the other, since they are suited for different situations [7]. In fact, current research on "quasi-static compilation" [27] shows that combining these two may yield excellent results.

In practice, static Java compilers are sometimes desirable over JIT compilers because they have many advantages [3, 5, 7]:

- Static compilation yields more robust deployed applications. On the one hand, a deployment JIT may be different from the development JIT, which can cause problems due to even slight differences in the virtual machine or library code. With static compilation, programs are compiled into native code allowing the developer to test exactly what is deployed. On the other hand, compilers have bugs. Crashes caused by static compiler bugs sometimes happen at compile time (unless the bug is the kind that generates bad code silently), while bugs in the JIT may cause crashes at program execution time, and some of them may only surface after a portion of the program has been executed many times. Moreover, if the program crashes due to a bug in either the compiler or the program itself, statically compiled code is much easier to debug because the run-time trace is more predictable.
- Static compilation provides better intellectual property protection. Native code is much harder to reverse-engineer than Java bytecode.
- Static Java compilers can perform resource intensive optimization before the execution of the program. In contrast, JIT compilers must perform analysis at execution time, thus are limited to simple optimizations that can be done without a large impact on the combined compile and execute time.
- Static compilation achieves greatly reduced startup cost, reduced memory usage, automatic sharing of code by the OS between applications, and easier linking with native code.
- Last but not least, static compilation is better suited for code certification than JIT compilation. It is significantly easier to achieve higher safety assurance by removing the compiler from the trusted computing base. There has been a lot of work done in this area [23, 21, 19] which mostly focuses on static compilation.

Regardless of the above advantages, there is an ongoing debate in the Java community on whether statically compiled implementations can meet the Java specification on dynamic features such as binary compatibility. Our paper presents a scheme that accommodates the seemingly contradictory goal of full Java compliance and static compilation, thus showing that binary compatibility can indeed be supported using static compilers. Following the inspiration of "quasi-static compilation" [27], this technique in practice can also be used together with other JIT compilation techniques to achieve an optimal balance point between static and dynamic compilation. Thus we believe this result is of interest to the general audience in the JVM community.

2 Background

Java binary compatibility requires that changes in certain aspects of a class C from version to version must not entail the recompilation of other classes that are clients of C. (Client classes of C are those that reference C in some way, such as by accessing members of objects of C, or by extending C.) For example, changing the order of methods and fields of a class or adding methods and fields to a class must not force recompilation of its clients.

Java virtual machines allow these kinds of changes to occur between releases of a class because references from one class to the fields and methods of other classes are made by symbolic names embedded in the class file. These references are transformed into addresses and offsets during the process of resolution.

However, static compilers that do not consider binary compatibility usually generate these offsets hard-coded, ahead of (link) time. This implies that changes to a class that affect the layout of fields and methods in the class could require all of its clients to be recompiled, since they contain hard-coded addresses and offsets based on the old layout. Failure to recompile all clients of a modified class can result in unexpected run-time behavior.

Current run-time compilers for Java have encountered similar problems. Taking the virtual method invocation as an example, binary compatibility is usually accomplished using run-time compilation techniques: Just-in-time compilers generate code for classes at run-time. During the run-time compilation, a virtual method invocation on an object of a loaded class can be safely compiled based on the determined *vtable* (virtual method table) of that class. However, a virtual method invocation on an object of a class which is not loaded yet cannot be handled in the same manner. In this case, the compiler emits special code which "stitches" the actual method invocation code lazily when it is executed.

For optimization purposes, JIT compilers often use guarded inlining (where the guard checks for the object type at run-time) to handle the scenario where the inlining is invalidated by further class loading. When such a scenario occurs, run-time compilation has to be performed. There has been also work on techniques for inlining virtual methods more efficiently [6, 16, 25].

Typically, static compilers use global or whole-program analysis [4] to do inlining or devirtualization. However, without the ability to perform compilation at run-time, they assume that no changes will be made to classes referred to by the compiled code. Hence, they do not comply with the JLS. The trade-off between binary compatibility (for full compliance with the JLS) and cross-class inlining is obviously an issue for static Java compilers. While our solution for binary compatibility does not directly support cross-class inlining, in cases when dynamic compilation may not be desirable due to various reasons discussed in section 1.1, an implementation would employ our solution together with other schemes that support inlining (but not binary compatibility) to achieve optimal results. For instance with quasi-static compilation [27] both pre-compiled native code and bytecode of classes are shipped together. When binary changes invalidate the pre-compiled native code, the VM falls back to compiling or even interpreting the bytecode. Similarly, a version of native code compiled with cross-class inlining can be shipped together with native code compiled using our approach. In the common case, when no changes to other classes are made, the inlined version of native code would be used for maximum efficiency. In cases when binary changes are detected, the system would fall back to running the version compiled without inlining but with support for binary compatibility, thus avoiding run-time compilation.

In summary, run-time compilers support binary compatibility with various run-time compilation techniques. However none of the existing static Java compilers provide support for binary compatibility, primarily because the high overhead negates much of the advantages of static compilation.

3 Static Compilation vs Binary Compatibility

In this section, we give examples to illustrate the concept of Java binary compatibility. We also show how the naïve application of the standard vtable approach for static compilation fails.

Consider the following program. Class Programmer defines two virtual methods, eat and hack. Class JavaProgrammer extends class Programmer, overrides those two methods, and defines a new virtual method study. The main method of class Manager creates instances of both the above classes and does some virtual method calls. Note that at run-time the variable Whoami contains an object of class JavaProgrammer, although its static class is Programmer.

```
public class Programmer {
  void eat  () { ... };
  void hack () { ... };
}
public class JavaProgrammer
     extends Programmer{
  void eat  () { ... };
  void hack () { ... };
  void study() { ... };
}
public class Manager {
  public static void main (String args[]) {
    // some code that runs for days...
    ...
    Programmer Tom = new Programmer();
    JavaProgrammer Jerry = new JavaProgrammer();
    Programmer Whoami = new JavaProgrammer();
    ...
    Tom.eat();
    Tom.hack();
    Jerry.eat();
    Jerry.hack();
    Jerry.study();
    ...
}}
```

The standard technique used in object-oriented programming language implementations supports virtual method dispatch by collecting the virtual methods of a class in a record called a *vtable*, and providing a pointer to this record in each object of the class. When the three classes above are compiled, the layout of the vtables of classes Programmer and JavaProgrammer is determined statically (Figure 1), and the code in class Manager is compiled to invoke virtual methods by accessing the corresponding entries in the vtables of these classes, reachable through the respective objects. Since the variable Whoami, declared of class Programmer, can be bound to an object of class JavaProgrammer, the layout of the vtable of JavaProgrammer must be consistent with that of Programmer, so that virtual method invocations can be compiled to use the same offset in the vtable for a given method of Programmer, regardless of the dynamic class of the object.

However, this vtable approach cannot be directly applied if we want to support binary compatibility. When changes are made to the binary of a class, the locations of method pointers in the vtable may change, which invalidates the offset information used to compile other classes. Even worse, vtables reachable through objects of the same static class may now have different layout, as illustrated in the following subsections.

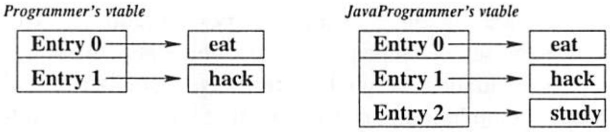


Figure 1: The vtables.

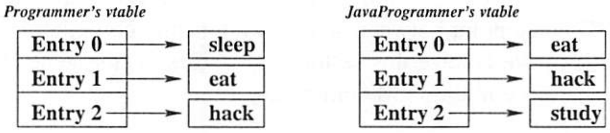


Figure 2: Scenario A: adding a method.

3.1 Scenario A: Adding a Method

Here we make a binary-compatible change to class Programmer by adding a new method sleep at the very beginning.

```
public class Programmer {
  void sleep() { ... }; // New Method!
  void eat  () { ... };
  void hack () { ... };
}
```

Now we recompile class Programmer only. The vtables for Programmer and JavaProgrammer are shown in Figure 2. The vtable layout of class Programmer has changed, and it is no longer consistent with the vtable layout of class JavaProgrammer. When these classes are loaded and Manager.main invoked, the code for Tom.eat will access the wrong entry in the vtable and end up calling method sleep. A similar problem occurs with Tom.hack. This is exactly the behavior shown by the current GCJ, which uses the standard vtable approach, and thus does not support binary compatibility.

Note that even if we had added the method sleep at the end of class Programmer, the problem still exists, because when we recompile class Programmer, method sleep will use entry 2 of the vtable. However the entry 2 of the vtable of class JavaProgrammer is already occupied by method study, and the invocation Jerry.study of Manager was compiled based on this.

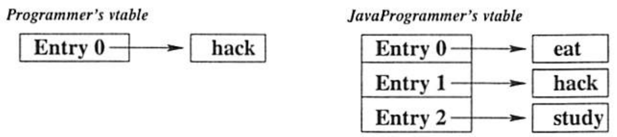


Figure 3: Scenario B: removing a method.

The observation here is that the vtable layout may change due to changes in the class. So we really should not have made any assumptions about the vtable layouts of `Programmer` and `JavaProgrammer` in `Manager`. Moreover, the information available at compile time is not sufficient for building the vtables, since classes in the same hierarchy may change, yet we still need to maintain consistency between a subclass and its superclass.

3.2 Scenario B: Removing a Method

Some source code modifications, such as removing a method from a class, are binary incompatible changes in the sense that other programs which work fine with the old binary may cease to function when linked with the evolved new binary due to the removal of the method. However, the safety of modern software systems demands that under no circumstances may an application crash. The JLS requires that, under the incompatible change in which a method is removed, the program should still run as long as the missing method is not used, and that an exception should be raised if code tries to invoke the missing method at run time.

Consider what happens when we remove the method `eat` from class `Programmer` in the original program and recompile it. The vtables for `Programmer` and `JavaProgrammer` are shown in Figure 3. Obviously, the vtable layout of class `Programmer` has changed, and it is no longer consistent with the vtable layout of class `JavaProgrammer`.

```
public class Programmer {
  // void eat  () { ... }; // Removed!
  void hack () { ... };
}
```

In this case, the correct behavior of a virtual method invocation `obj.eat` in any old binary depends on the static class of the object `obj`. If the static class of `obj` is `JavaProgrammer`, the method invocation works fine, as if no change had been made. However, if the static class of `obj` is `Programmer`, a `NoSuchMethodError` exception should be thrown when the method is invoked, even if `obj` actually contains an object of class `JavaProgrammer` which defines method `eat`.

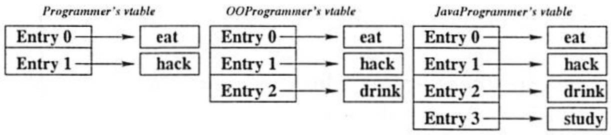


Figure 4: Scenario C: changing class hierarchy.

The standard vtable approach fails in this case as well. The invocation `Tom.eat` in `Manager` will call the wrong method `hack`, while `Tom.hack` will have implementation-dependent results, since it uses a pointer located outside of the actual vtable.

The observation here is that we need to gracefully handle incompatible changes by raising exceptions at run time. Of course, we still need to keep in mind the consistency of vtables.

3.3 Scenario C: Binary Change at Run Time

Some static compilers (e.g. BulletTrain) perform dependency analysis before executing a Java program, and attempt to recompile if inconsistency is detected. In the cases of scenarios A and B, these compilers would have refused to run `Manager`, or attempted to recompile it automatically. The problem is that this behavior is not only non-compliant with the binary compatibility requirements of the JLS (which intends to solve these issues without recompilation of the client classes of the changed class), but also that this dependency analysis cannot always be done statically.

A simple example which presents a challenge to the static dependency analysis scheme is reflection. Using reflection, a program can load arbitrary class files which are not known at compile time. This means that the static analysis may not work out all the dependencies. Even if reflection is not a concern, binary changes may occur at run time after some classes are already loaded and executed. In these cases, recompilation is not possible. Here we use another binary-compatible change, namely the insertion of a new class into the class hierarchy, to demonstrate the problem.

```
public class OOProgrammer
     extends Programmer { // New Class!
  void drink() { ... };
}
public class JavaProgrammer
     extends OOProgrammer{ // New Hierarchy!
  void eat  () { ... };
  void hack () { ... };
  void study() { ... };
}
```

Based on the original program, before we make any changes, suppose class `Manager` (but not `JavaProgrammer`) is already loaded and being executed. During the execution of the first line of `main`, we insert a new class `OOProgrammer` between `Programmer` and `JavaProgrammer`. We compile `OOProgramer` and recompile `JavaProgrammer`. The vtables are shown in Figure 4. Clearly, the vtable layout of `JavaProgrammer` has changed. The line for `Jerry.study` in `Manager` is going to call method `drink` which happens to reside in the entry 2 of `Jerry`'s vtable after the change.

The lesson is, binary changes may occur after some classes are loaded. Static dependency analysis and recompilation are sometimes not only undesirable, but unaffordable. Reflection is an additional complication.

4 Our Approach

In this section we present our solution for static compilers to support Java binary compatibility. Table 1 and Table 2 summarize all the binary changes to classes and interfaces specified in Chapter 13 of the Java language specification [12]. Compatible changes are marked with "+" and incompatible changes are marked with "−". We present the solutions for these changes in the coming subsections. In these tables, SEC x means the solution is presented in Section x. The symbol √ is only used for binary compatible changes. It means that the solution is trivial and requires no change to the existing implementation. The numbers associated with each binary change are used for cross-referencing in later sections.

4.1 Virtual Methods

The major difficulty in supporting Java binary compatibility is the handling of virtual methods. In this section, we present our idea in a simplified setting where only virtual methods under single inheritance are considered. Temporarily putting aside other features (e.g. modifiers) makes it easier to understand our solution. Extending this solution to work for static methods and constructors is trivial. All the other language features can be supported with simple extensions which we present later.

4.1.1 Idea

From the lessons we learned in Section 3, we know that even though we want to compile classes ahead of time, we cannot afford to build the vtables statically. The information we get during the ahead-of-time compilation

No.	+/−	Binary Change to Class	Solution
1	+	adding(overriding) method/constructor (without modifier change)	SEC 4.1
2	+	changing hierarchy preserving super(s)	SEC 4.1
3	+	adding field (without modifier change)	SEC 4.2
4	+	abstract → nonabstract	√
5	+	final → nonfinal	√
6	+	nonpublic → public	√
7	+	allowing more access to member	√
8	+	final field → nonfinal field	√
9	+	adding/deleting transient modifier	√
10	+	changing formal parameter name of method/constructor	√
11	+	abstract method → nonabstract	√
12	+	final method → nonfinal; nonfinal static meth → final static	√
13	+	changing synchronized modifier	√
14	+	changing throws clause	√
15	+	changing method/constructor body	√
16	+	adding method/constructor that overloads existing one	√
17	+	changing static initializer	√
18	−	removing method/constructor	SEC 4.1
19	−	removing field	SEC 4.2
20	−	changing hierarchy without preserving super(s)	SEC 4.4.1
21	−	noncircular hierarchy → circular hierarchy	SEC 4.4.1
22	−	nonfinal → final	SEC 4.4.1
23	−	nonabstract → abstract	SEC 4.4.1
24	−	nonfinal virtual method → final virtual	SEC 4.4.2
25	−	restricting access to member	SEC 4.4.2
26	−	nonfinal field → final field	SEC 4.4.2
27	−	static ↔ instance member	SEC 4.4.2
28	−	nonabstract method → abstract	SEC 4.4.2
29	−	public → nonpublic	SEC 4.4.2
30	+/−	adding(overriding) method (with modifier change)	SEC 4.4.3
31	+/−	adding field (with modifier change)	SEC 4.4.3
32	+/−	changing signature	SEC 4.4.3
33	+/−	about native methods	SEC 4.4.3

Table 1: Java binary compatibility summary: classes.

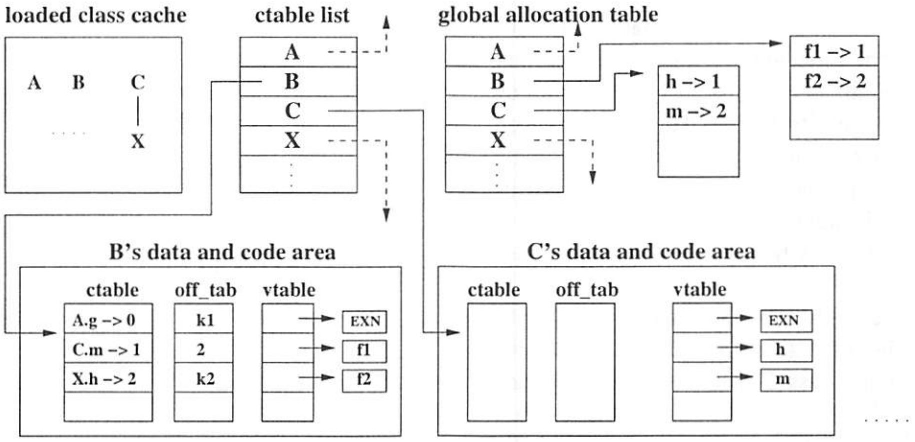


Figure 5: Our solution.

No.	+/−	Binary Change to Interface	Solution
34	+	changing hierarchy preserving super(s)	SEC 4.3
35	+	nonpublic → public	√
36	+	adding/deleting transient modifier	√
37	+	changing formal parameter name of method	√
38	+	changing synchronized modifier	√
39	+	adding method that overloads existing one	√
40	−	removing member	SEC 4.3
41	−	changing hierarchy without preserving super(s)	SEC 4.4.1
42	−	public → nonpublic	SEC 4.4.2
43	+/−	adding field	SEC 4.2 SEC 4.4.3
44	+/−	adding method	SEC 4.3 SEC 4.4.3
45	+/−	changing signature	SEC 4.4.3

Table 2: Java binary compatibility summary: interfaces.

is not sufficient to determine the vtable layout. Besides, we need to handle our compilation carefully so that we can detect binary-incompatible changes and emit error messages gracefully.

We solve this problem by building vtables during class loading. Once loaded, a class is considered fixed. Further changes to this class can be ignored, according to the JLS. Thus we can safely determine the layout of the vtables.

A minor complication is that vtable layouts have to be consistent between a superclass and a subclass. In other words, a method m is located at the same position in the vtable of a subclass as in the vtable of a superclass. Luckily, the loading of a superclass precedes the loading of a subclass, which makes it possible to construct the vtable of the subclass based on the vtable layout of the superclass. In our solution, we maintain this consistency with the help of a global allocation table which reflects the layout of the vtables of all the loaded classes. During the loading of a class, we check the global allocation table to learn the vtable layout of the superclass. Then we follow the layout of the superclass and construct the class's vtable by appending fresh entries at the end. We also record the newly determined layout in the global allocation table so that any subclasses can access it.

The problem now is how to statically compile a virtual method invocation when the vtable layout is not determined statically. We handle this by introducing an extra level of indirection by compiling virtual method invocations to fetch an offset table entry before accessing the vtable. The offset table maps a virtual method to the offset of the method in the vtable. Its entries are filled in at run time when the corresponding class is loaded.

The idea of our approach is shown in Figure 5. To enable this approach, we need to make changes to both the compiler and the class loader.

Compiler Every class is statically compiled to contain a customizing table (ctable) and an offset table (off_tab). The size of the ctable is proportional to the number of distinct external method invocations in the class. For each distinct external method referenced in the code of the class, there is a corresponding entry in the ctable. In a class C, if an external method f is invoked on both an object of a class A and an object of its subclass B, then both $A.f$ and $B.f$ will appear in C's ctable. A ctable entry maps an external method to a unique natural number. This natural number is the offset of the entry for the external method in the offset table. The offset table entries are filled in incrementally at run time according to the information in the global allocation table. A virtual method invocation is compiled to go through the corresponding offset table entry before accessing the vtable.

Class loader The class loader has to maintain the global allocation table, the offset tables, and the vtables during class loading. When a class X is loaded, the class loader constructs the vtable for X based on the vtable layout of the superclass of X, which is specified in the global allocation table. Here we reserve the entry 0 to point to some special exception code. Once the vtable is constructed, the class loader registers the vtable layout in the global allocation table. This information is also propagated to the offset tables of the loaded client classes of X. In this step it is possible that the offset table of a certain class A contains an entry for a method m of class X, while m does not actually exist in this newly loaded class X. This means that there must have been some binary incompatible changes (e.g. m was removed from X, while A was compiled with an old version of X). In this case, the class loader puts the special offset 0 in the corresponding offset table entry. The entry 0 of a vtable always points to some special code that would raise proper exceptions when the method m is invoked.

Example Consider what happens at run time when executing a virtual method invocation `o.m`, where `o` is of static class `C`. Suppose this method invocation appears in the body of class `B`. The statically determined ctable of class `B` designates an offset k for the method `m` of class `C`. In our scheme, `o.m` is compiled to access the entry k of class `B`'s offset table for a new offset k'. This new offset k' is copied from the global allocation table during class loading. It is the actual offset of the method `m` in the vtable of class `C`. Although the dynamic class of object `o` could be a subclass of `C`, it is safe to use the offset k' to access the vtable of object `o` for invoking the method, because we have arranged the vtables of a superclass and its subclasses to be consistent.

Correctness The correctness of our solution for virtual methods is based on the following observations: the global allocation table provides a correct view of all the vtables of the loaded classes; the vtable of a subclass is consistent with the vtable of its superclass; all offset tables are consistent with the global allocation table; and a virtual method invocation cannot be executed before the class of the receiver object is loaded. We refer interested readers to our internal report [31] for a formal development and its soundness proof.

4.2 Fields

The support for fields can be separated into three categories: support for private (or even protected) fields, support for non-private fields, and support for various kinds of access modifiers.

Private fields can only be referenced from within the defining class. Thus they do not require any special care for binary compatibility. Protected fields can only be referenced from the defining class or a subclass. In this case, the loading of the referencing class cannot precede the loading of the defining class. We can completely patch the references to these fields during class loading because by that time the layout of these fields is already determined. By doing this, we do not have to go through the extra indirection of the offset tables.

Changing non-private fields may affect other classes that depend on them. A similar technique as we used for methods can be used here, though it may be relatively less efficient. However, it is generally good software engineering practice to limit the use of non-private fields. Using non-private fields is also discouraged in the Binary Compatibility chapter of the JLS; to quote from Section 13.4.7 of JLS [12], "Widely distributed programs should not expose any fields to their clients." In fact, non-private fields (especially as part of public APIs) seem to be quite rare. To the authors' knowledge, they are almost non-existent in the standard Java libraries. We believe that the inefficiency here will not have much impact in practice.

Nevertheless, the handling of removed fields is tricky. Unlike calling a method, the trick with reserving the 0-offset entry will not work in this case because accessing it as a field will not raise any exceptions. Using a run-time check for every field access to determine whether the offset for a field is valid has too great a cost in performance. Our solution is, instead of detecting missing fields lazily, to raise exceptions at class loading time when trying to fill in an offset table entry for a field, if the corresponding information is not in the global allocation table. Note that this solution does not obey the JLS on the particular aspect that exceptions of a missing field should be raised lazily. For full compliance with the JLS, one possible solution is to fill in the offset table entry of the missing field with some special offset which triggers an OS trap when accessed. A similar technique is introduced by Joisha *et al.* [17] for the IBM Quicksilver quasi-static compiler [27] for a different purpose, namely to trigger the "stitching" (or linking) operations.

More surprisingly, adding a field is not always a compatible change if changes of modifiers are involved. We discuss this peculiarity in Section 4.4.3 together with other modifier changes.

4.3 Interfaces

The common practice in supporting Java interfaces is to use interface tables (itables); we refer the reader to the work of Alpern *et al.* [2] for a discussion of the prior implementation techniques for interface dispatch and an efficient implementation of Java interfaces. For each interface that a class implements, there is a corresponding itable which contains all the methods declared in the interface. At run time, an interface method invocation would involve looking up the itable by interface name, fetching the address of the interface method from a fixed offset, determined at compile time, and invoking it. The itable look-up mechanism provides natural support for binary compatibility; however, if the method layout of an itable may change, it would be wrong to use a fixed offset to access an interface method.

Fortunately, this is exactly the same problem that we solved for virtual methods and vtable dispatch. All we have to do is make sure interface method invocations go through the offset table, and fill in the offset table incrementally at class loading time once the itable layout is determined.

4.4 Other Changes

All the other binary changes specified by the JLS can be supported by making simple extensions to the techniques discussed so far. They happen to all be incompatible changes, and fall into the following categories. (The numbers at the beginning of the bullets are used for cross-referencing with the entries in Table 1 and Table 2.)

4.4.1 Checking constraints

The incompatible changes in this category are handled by maintaining constraints either explicitly or implicitly, and checking them against the loaded classes during class loading. When any of the constraints are violated, exceptions are raised (`VerifyError`, `ClassCircularityError`, `InstantiationError`, etc).

- (20,41) When compiling a class, we add a constraint for every upward cast indicating the expected inheritance relationship. During execution, we have the system maintain all the constraints specified by the currently loaded classes. When a class is loaded, we check its constraints against the loaded class hierarchy. We also check the newly loaded class against the constraints maintained by the system.
- (21) If the class hierarchy becomes circular due to incompatible changes, we can detect it during class loading.
- (22) If a class Sub inherits a nonfinal class $Super$, and $Super$ is changed to be final, we can detect it during the loading of class Sub.
- (23) Abstract classes cannot be used to create instances. Similar to what we did for upward casts, we add a constraint for every instance creation indicating that the class being instantiated cannot be an abstract class. These constraints are checked during class loading.

4.4.2 Tagging and exception handling

Most modifier-incompatible changes can be handled by tagging the global allocation table entries with modifiers (e.g. access control, readable/writable, instance/static, etc.). In the offset tables, the entries are tagged with the expected modifiers, too. During class loading, the class loader decides whether the modifiers are compatible, and fills in an offset table entry with the registered offset only if they are.

While we could use offset 0 for all kinds of error handling, it is usually preferred to raise different exceptions on different incompatible changes. To achieve this, we can reserve more entries in the vtables and other data structures (e.g. itables) for various exception code. If a certain access is denied according to the global allocation table, the offset of the corresponding exception entry is used.

- (24) Final virtual methods cannot be overridden. During class loading, a subclass studies the vtable layout of its superclass. A tag in the global allocation table indicates whether a virtual method is final. If a final virtual method is overridden, `VerifyError` exception is raised immediately.
- (25) If a binary change restricts access to a member, the tag in the corresponding global allocation

table entry can be used to decide whether an access is granted. If an access attempt to a field is denied, `IllegalAccessError` exception is immediately raised. If an access attempt to a method or constructor is denied, the offset of the exception entry is used.

- (26) If a field that was not final is changed to be final, then it can break compatibility with preexisting binaries that attempt to assign new values to the field. Our solution is to tag the final field with read-only access in the global allocation table. If some offset table is expecting the field to be tagged with writable access, `IllegalAccessError` exception is raised.

- (27) Changing the `static` modifier of members could raise exceptions when the member is accessed. Static members and instance members are tagged differently in the global allocation table and the offset tables. In the case of field modifier mismatch, `IncompatibleClassChangeError` exception is raised; while in the case of modifier mismatch of other members, the entry in the table is filled with the offset for code raising the exception.

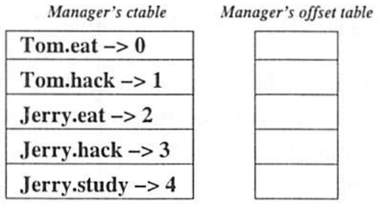
- (28) An abstract method cannot be invoked. If a subclass S inherits a method f defined in the old binary of a superclass C, and C is changed by declaring f as an abstract method, invoking f on an object of S is going to raise an exception (`AbstractMethodError`). Our solution is, when constructing the vtable of a class C that declares an abstract method f, to put a pointer to the exception code in the entry of f. A subclass that does not define f inherits the exception code, while a subclass that defines f works as if no change is made.

- (29,42) The members of nonpublic classes cannot be accessed from outside the package. Having access tags in the corresponding global allocation table entries solves this problem. Similar observations apply to interfaces.

4.4.3 Miscellaneous

- (32,45) Changing a signature has the combined effect of removing the old member and adding a new one.

- (33) Adding or deleting a native modifier is considered compatible. The support for this is trivial. Other changes related to native methods are beyond the scope of the JLS.



Manager's ctable	*Manager's offset table*
Tom.eat –> 0	
Tom.hack –> 1	
Jerry.eat –> 2	
Jerry.hack –> 3	
Jerry.study –> 4	

Figure 6: The ctable and offset table of `Manager`

- (30,31,43) Adding a field/method is a compatible change, except in the following cases.

    1. The new field/method shadows/overrides an old one, and the new field/method is less accessible than the old one.
    2. The new field/method shadows/overrides an old one, and the new field/method is a static (instance) member but the old one is an instance (static) member.
    3. The new field in an interface may shadow a field in other classes.

    These cases are handled as (25), (27) and (26).

- (44) Adding methods to interfaces is listed as a binary compatible change in the JLS. However, it MAY break compatibility with pre-existing binaries [8]. Based on our support for interfaces in Section 4.3, together with the technique described for (28), we can build the itable of an interface with pointers to specific exception code. If a class which implements the interface does not define all the interface methods, the exception code is inherited.

5 Example Revisited: Programmer & Manager

In our solution, the class `Manager` is compiled to contain a ctable and an offset table (Figure 6). The ctable maps distinct external methods used in the class to unique offsets. The offset table is initially empty, and is filled in incrementally at run time with the offsets of these external methods.

5.1 Scenario A Revisited: Adding a Method

In Scenario A we added a new method `sleep` at the very beginning of `Programmer`. Our solution does not require recompilation of `Manager`, because `Manager` does not make any assumptions about the vtable layouts of `Programmer` and `JavaProgrammer`.

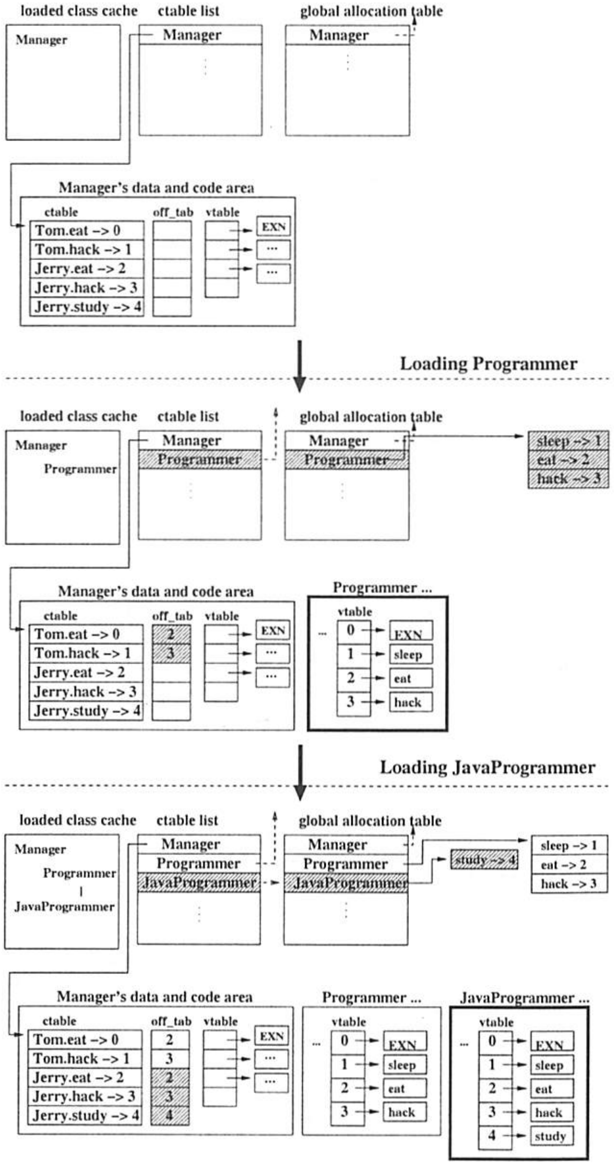


Figure 7: Scenario A revisited: adding a method.

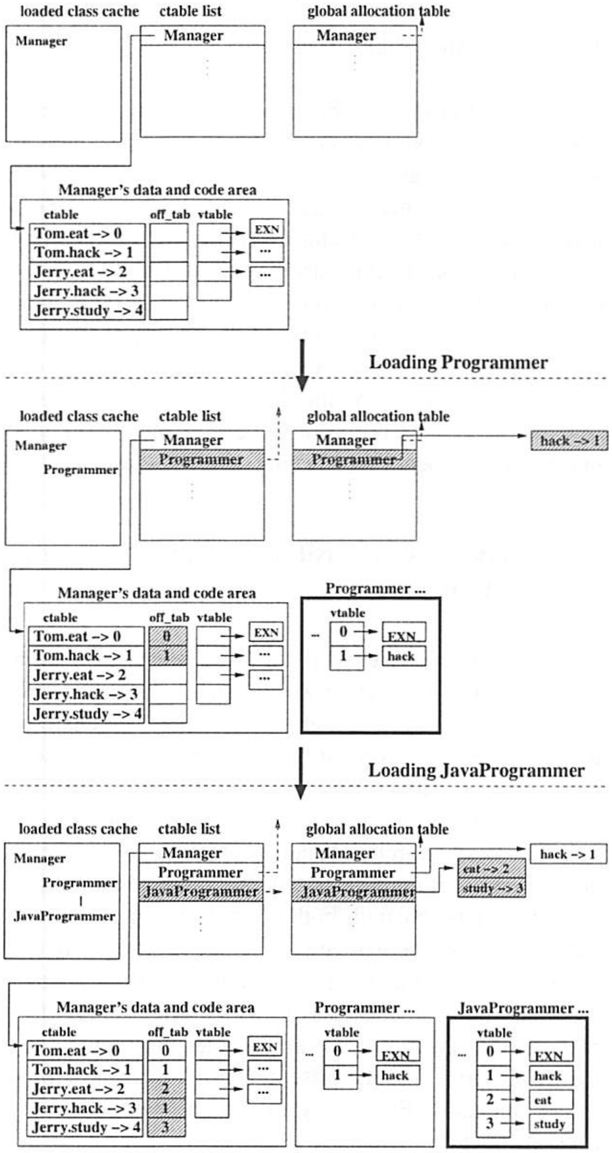


Figure 8: Scenario B revisited: removing a method.

Figure 7 illustrates what happens during the class loading of Programmer and JavaProgrammer. When Programmer is being loaded, its vtable is constructed on the fly so that any binary changes before class loading can be taken into account. The layout of this newly constructed vtable is registered in the global allocation table entry of Programmer. This layout information is used to fill in the relevant offset table entries of Manager.

When JavaProgrammer is loaded later (being a subclass of Programmer, its loading cannot precede that of Programmer), its vtable is constructed following the layout requirements specified by the global allocation table entry of Programmer. By doing this, the consistency of the vtable layouts between a subclass and its superclass is maintained. Once this is done, the layout information is propagated accordingly to the offset table entries of Manager. Using the offset specified in the corresponding offset table entry, a virtual method invocation will successfully get through.

5.2 Scenario B Revisited: Removing a Method

In Scenario B, we removed the method eat from class Programmer. Although this is an incompatible change, the program is still supposed to run as long as the removed method is not invoked. In class Manager, if we execute a method invocation Jerry.eat, the program will execute as usual. However, if we execute Tom.eat, an exception should be raised.

In our solution (Figure 8), after class Programmer is loaded, there would be no information about method eat in the global allocation table. However in Manager, which expected a method eat from class Programmer, there is an offset table entry which expects an offset for method eat. In this case we put 0 in the offset table entry. In the actual vtable of any class, the entry at offset 0 is always set to point to some specific code that would raise proper exceptions when invoking the Tom.eat method at run time. However, things will be as usual if Jerry.eat is executed, because eat does occur in the global allocation table entry of JavaProgrammer.

5.3 Scenario C Revisited: Binary Change at Run Time

Our solution works even if binary changes happen at run time (Figure 9). Here we illustrate why this is true using the example in scenario C, in which we changed the class hierarchy compatibly by inserting a new class. In scenario C, after class Manager is loaded and being executed, we add a new class OOProgrammer into the old class hierarchy. When object Tom is being created, class Programmer is loaded. Then, when object Jerry is being created, both class OOProgrammer and class JavaProgrammer are loaded. The vtable layouts of Programmer, OOProgrammer and JavaProgrammer are determined during class loading. All of this information is registered in the global allocation table. Using this information, the offset table of Manager is filled in incrementally. Eventually, the virtual method invocations will get through.

6 Algorithm

We have separately talked about how to support various binary compatibility issues in Section 4. In this section, we take a different view and present roughly the overall algorithm of our solution. The algorithm consists of two parts: the *compiler* part and the *class loader* part.

6.1 Compiler

To support binary compatibility, the major difference in the compilation is that the metadata structures (e.g. vtable, itable, field record, etc) of classes and interfaces are not fixed.

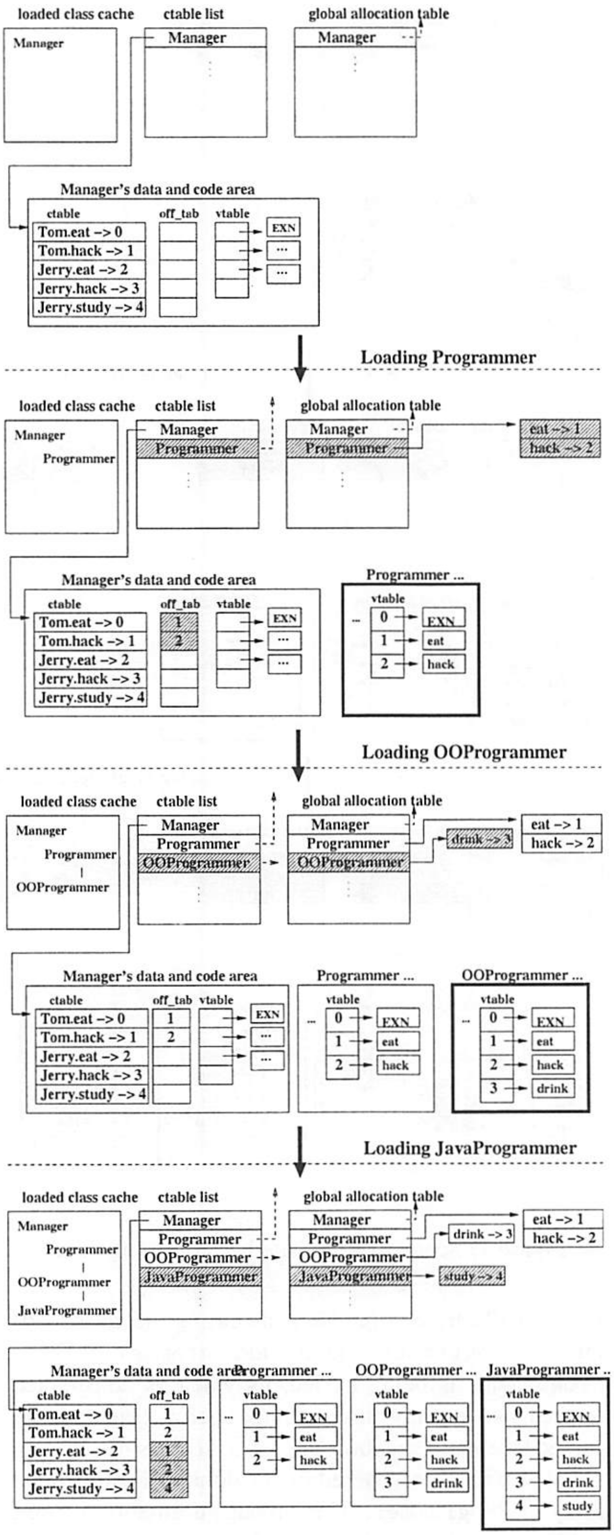


Figure 9: Scenario C revisited: changing class hierarchy.

1. *Create ctable and offset table for every class being compiled.* The ctable maps external references

(including references to various kinds of members) to unique offsets to the offset table. The offset table entries are tagged with the expected modifiers of the members. The contents of the offset table entries are blank. They are to be filled in incrementally at run time when the corresponding class is loaded. It is guaranteed that an offset table entry will be filled in before it is used, because no access to a class can be made before the class is loaded.

2. *Compiling external references.* Accesses to external references are compiled to go through the offset table. The object code fetches an offset from the offset table, and uses it to access the corresponding metadata structure.
   Taking virtual method invocation as an example, if an object `o` is of static class X, then a virtual method invocation `o.m()` that appears inside class C would be compiled as follows (where the final `o` is the self pointer):

   ```
   let off_m = lookup(ctable, "X", "m")
    in o.vtable [off_tab[off_m]] (o)
   ```

   Here `ctable` is the ctable of class C, `off_tab` is the offset table of class C. Class C's ctable entry for the virtual method m of class X is fixed. The lookup can be performed at compile time, so that at run time we can fetch the vtable offset directly from a certain offset table entry.

6.2 Class Loader

The class loader needs to maintain the related data structures such as the global allocation table, offset tables, and metadata structures like the vtable. It also has to check for various constraints during class loading. Here is what happens when a class C is loaded by the class loader. Loading interfaces is handled similarly.

1. *Loading superclasses.* Recursively load all the superclasses of C, if they are not loaded already.

2. *Check for constraints related to the class hierarchy.* In this step we only need to check things related to class C. Refer to Section 4.4.1 for details.

3. *Create a global allocation table entry for class C (T_c).* This T_c maps every member to its position information in the corresponding metadata structure (e.g. vtable), together with the modifier tags. Static members are easy to deal with, but special attention must be paid when mapping instance members, because we have to do it in such a way that it is consistent with the global allocation table entries of C's superclasses.
   In order to do this, we have to check the global allocation table entries of all the superclasses of C (recursively, they are already consistent with each other). Note that this data is not copied from the global allocation table entries of C's superclasses to the entry of C, because that would be space-inefficient. This is also when we make sure final methods are not overridden. After that, we construct the mappings of C's fresh instance members (those members defined in C but not in any superclasses of C). Here we can only use those offsets which are not yet used in any superclasses of C. In particular some offsets (e.g. 0) are reserved for incompatible change exception handling and cannot be used to map members to.

4. *Create the class data structures (e.g. vtable) of class C.* We do this according to the layout specified in the global allocation table entry C and the entries of C's superclasses. If the global allocation table maps a member to position k, then the data for the actual member is put at position k of the corresponding data structure. Beside those specified by the global allocation table, we also need to fill in the reserved entries with pointers to particular exception code.

5. *Fill in currently loaded classes' offset table entries that correspond to the members in C.* We do this with the help of the global allocation table entries of C and its superclasses. This step is done by iterating over the ctable list. An inverse ctable list would probably help to improve efficiency.
   Here we may find out that other classes might be expecting a non-existent member from C, or the expected access is not granted by comparing the modifier tags. In these cases we put offsets of exception code (e.g. 0) in the offset tables of those classes. However, as we discussed in Section 4.2, removed fields cannot be handled in the same manner. We raise an exception immediately when trying to fill in the offset table entry for that removed field.

6. *Fill in the offset table of class C.* This is done in the same manner as the last step. For C's offset table, we fill in those entries that correspond to members expected from the loaded classes. The entries for members expected from not-yet-loaded classes are left to be filled in later.

7. *Add the information of class C to the loaded class cache.* Also add the class hierarchy constraints demanded by class C to the set of constraints maintained by the system.

```
movl _ZN4java4lang6System3outE, %eax ;object
movl (%eax), %edx                    ;vtable
movl %eax, (%esp)                    ;this
movl _CD_C+4, %eax                   ;argument
movl %eax, 4(%esp)
call *116(%edx)
```

direct dispatch (vtable)

```
movl _ZN4java4lang6System3outE, %ecx ;object
movl _CD_C+4, %eax                   ;argument
movl (%ecx), %edx                    ;vtable
movl %ecx, (%esp)                    ;this
movl %eax, 4(%esp)
movl otable+4, %eax                  ;offset
call *(%eax,%edx)
```

indirect dispatch (offset table)

Figure 10: Sample method invocation code

8. *Extend the ctable list with a pointer to the ctable of class C.*

7 Implementation and Performance Evaluation

For ease of presentation, we have used vtable entry numbers in the offset table entries in the examples. In an actual implementation, a "processed offset" can be used instead. This "processed offset" is the vtable entry number multiplied by the size of a vtable entry (size of a pointer to code). Thus we transferred some of the burden from run time to compile time.

Bryce McKinlay implemented part of our solution for GCJ. His implementation so far provides full support for virtual methods and partial support for interface methods. The support for fields is still work in progress. This implementation is to be included in the future GCC release 3.1.

When compiling with the `-O2` optimization flag, it turns out that our new scheme generates one more assembly instruction for each virtual method invocation. Figure 10 shows the result of compiling a virtual method invocation. In our indirect dispatch scheme with the offset table involved, the code fetches the offset from the corresponding offset table (`otable`) entry, and adds it to the vtable pointer before calling the method. In contrast, the original direct vtable dispatch scheme generates code which adds the offset of the method to the vtable pointer and calls it. When the same call occurs in a loop (or in succession), the compiler moves the otable load out of the loop, so the overhead is reduced.

Benchmark	Direct	Indirect	Unit	Ratio (%)
Same:Instance	34.59	34.84	ns/call	99.27
Other:Instance	38.55	37.29	ns/call	103.39
Crypt (A)	7.26	7.27	s	99.82
HeapSort (A)	2.14	2.15	s	99.67
Series (A)	46.90	47.62	s	98.49
Crypt (B)	48.41	48.49	s	99.82
HeapSort (B)	14.69	14.67	s	100.14
Series (B)	485.87	493.12	s	98.53
AlphaBeta	24.76	25.04	s	98.89
MonteCarlo	32.89	32.64	s	100.77
Euler	327.55	328.65	s	99.66
RayTracer	45.21	44.81	s	100.90
MolDyn	518.09	529.13	s	97.91

Table 3: Java Grande 2.0 benchmarks

Our tests are based on the Java Grande 2.0 benchmarks [9] (the current version of GCJ cannot compile the SPECjvm98 benchmark suite). All results were obtained on a DELL Precision 410 workstation running Red Hat Linux 7.1. The machine has 512MB of main memory and 500MHz Pentium III processor with 512KB of cache. The average results over 3 rounds of tests, using dynamic linking with `-O2` flag turned on for both the direct vtable dispatch scheme of GCJ (Direct) and our indirect offset table dispatch scheme (Indirect), are shown in Table 3. In the "Ratio (%)" column, numbers less than 100 indicate performance slowdown using our scheme, while numbers greater than 100 indicate performance speedup.

The first two benchmarks are taken from benchmark suite section 1 (Low Level Operations). They test the performance of invoking virtual (instance) methods on an object of the same class and of another class. These two benchmarks perform a large number of iterations over 17 method invocations; every invoked method simply increases a global static counter. The result indicates that much of the offset table overhead was optimized away in these cases. Somewhat surprisingly the performance on "Other:Instance" was improved, possibly due to the different instruction scheduling.

The rest of the benchmarks (IDEA Encryption, Heap Sort, Fourier Coefficient Analysis, Alpha Beta Search, Monte Carlo Simulation, Computational Fluid Dynamics, 3D Ray Tracer, and Molecular Dynamics Simulation) are chosen from Sections 2 (Kernel) and 3 (Large Scale Applications) of the benchmark suite. We chose those with the most method invocations involved. Some of them are run on different data sizes (A/B). The performance penalty is on average less than 2%. Again we see some performance speedup in the test cases.

Another interesting observation is on the size of the object files. The new indirect dispatch scheme for binary compatibility puts extra offset tables in the object files. However, the vtables are no longer needed. When testing with the Java Grande 2.0 benchmarks, it turned out that the object file size using the new scheme is on average 1% less than using the standard vtable dispatch scheme.

8 Related Work

Joisha *et al.* [17] use an indirection table to enable efficient sharing of executable code for the IBM Quicksilver quasi-static compiler [27]. Besides increasing reusability of binary code, their solution also provides some support for binary compatibility. When binary changes (especial compatible ones) to a class C are detected during class loading, the class C is recompiled without requiring any changes to the loaded client classes of C, because all stitching (or linking operations) are performed on the indirection table. This stitching, or the operation which fills in the indirection table, happens incrementally the first time any single entry is used during the execution of the program. To enable this operation, they use some special offsets to trigger "traps" in the OS. When the program tries to access the memory using these offsets, the trap handler takes care of filling in the indirection table entry and resuming the program execution. Since the major concern of their paper is the sharing of code images, they do not explicitly address the handling of various binary incompatible changes.

In contrast, our solution does not require using any dynamic compilation techniques. Unlike the approach taken by Joisha *et al.*, we handle the problem of binary compatibility by building vtables and other class data structures not during compilation but during class loading. A global allocation table is introduced to help maintain the consistency of table layout between superclasses and subclasses. We also introduce an offset table for every class. These offset tables are filled in with the help of the global allocation table during class loading as soon as the referenced class is loaded. The statically compiled code (e.g. for method invocation) uses the offset tables to access the corresponding class data structures (e.g. vtables). Due to the observation that an external reference cannot be executed before the referenced class is loaded, going through the offset tables is guaranteed to be safe. Thus our approach does not rely on OS-dependent trapping mechanisms to trigger the linking process at runtime. However, similar trapping mechanisms can be used to handle missing fields in cases when full compliance with the JLS is important. Lastly, because we fill in all related offset table entries during the loading of a class, we can check for various binary incompatible changes. Our paper presents a detailed discussion of all the binary changes, including both compatible and incompatible, defined by the JLS.

9 Conclusion

We have presented a scheme which uses static compilation to support Java binary compatibility. All of the binary compatibility requirements in the Java Language Specification are supported with the same set of simple techniques. Binaries changed in a compatible manner can link successfully with pre-existing binaries that previously linked without error. Incompatible changes raise various run-time exceptions accordingly. Our implementation shows that this approach is fairly efficient and has the potential of being applied to real systems.

10 Acknowledgments

We want to specially thank Bryce McKinlay for implementing the indirect dispatching scheme for GCJ and his inspiring discussion on related issues, and Manish Gupta for keeping us informed about the state-of-the-art in dynamic Java compilers. We also thank many posters on the GCJ mailing list (java@gcc.gnu.org) for answering our questions about the GCJ internals, Greg Collins and Andrew McCreight for valuable comments on an early version of our paper, and the anonymous reviewers.

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM System Journal*, 39(1), February 2000.

[2] B. Alpern, A. Cocchi, S. Fink, D. Grove, and D. Lieber. Efficient implementation of Java interfaces: Invokeinterface considered harmless. In *Proc. 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 108–124, 2001.

[3] P. Bothner. A GCC-based Java implementation. In *IEEE Compcon 1997 Proceedings*, pages 174–178, February 1997.

[4] D. Chambers, C. Dean, J. Grove. Whole-program optimization of object-oriented languages. Technical Report TR-96-06-02, Dept. of Computer Science and Engineering, University of Washington, 28, 1997.

[5] D. Chase, R. Hoover, and K. Zadeck. BulletTrain technology white paper. `http://www.naturalbridge.com/`, 2001.

[6] D. Detlefs and O. Agesen. Inlining of virtual methods. In *ECOOP'99, LNCS 1628*, pages 258–278, 1999.

[7] O. P. Doederlein. The Java performance report – part IV (static compilers). `http://www.javalobby.org/members/jpr/`, August 2001.

[8] S. Drossopoulou, D. Wragg, and S. Eisenbach. What is Java binary compatibility? *ACM SIGPLAN Notices*, 33(10):341–358, 1998.

[9] Edinburgh Parallel Computing Centre. The Java Grande forum benchmark suite. `http://www.epcc.ed.ac.uk/javagrande/`, 2001.

[10] Excelsior, LLC. Excelsior JET. `http://www.excelsior-usa.com/jet.html`, 1999.

[11] I. R. Forman, M. H. Conner, S. H. Danforth, and L. K. Raper. Release-to-release binary compatibility in SOM. In *Proc. 1995 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 426–438, Oct. 1995.

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification (Second Edition)*. Addison-Wesley, 2000.

[13] IBM Corporation. IBM's system object model (SOM): Making reuse a reality. First Class, a bimonthly publication of the Object Management Group, October 1994.

[14] IBM Corporation. IBM VisualAge for Java. `http://www.software.ibm.com/ad/vajava/`, 1998.

[15] Instantiations, Inc. Jove, optimizing native compiler for Java technology. `http://www.instantiations.com/jove/product/thejovesystem.htm`, 2000.

[16] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. A study of devirtualization techniques for Java Just-In-Time compiler. In *Proc. 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 2000.

[17] P. G. Joisha, S. P. Midkiff, M. J. Serrano, and M. Gupta. A framework for efficient reuse of binary code in Java. In *Proc. 15th ACM International Conference on Supercomputing*, pages 440–453, New York, June 2001.

[18] A. Krall and R. Grafl. CACAO — A 64-bit JavaVM just-in-time compiler. *Concurrency: Practice and Experience*, 9(11):1017–1030, 1997.

[19] C. League, V. Trifonov, and Z. Shao. Type-preserving compilation of featherweight Java. In *Proc. 8th Foundations of Object-Oriented Languages Workshop*, London, January 2001.

[20] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification (Second Edition)*. Addison-Wesley, 1999.

[21] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. In *Proc. 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–97, Jan. 1998.

[22] NaturalBridge, Inc. BulletTrain optimizing compiler and runtime for JVM bytecodes. `http://www.naturalbridge.com/`, 1996.

[23] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proc. 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 333–344, 1998.

[24] H. Ogawa, K. Shimura, S. Matsuoka, F. Maruyama, Y. Sohda, and Y. Kimura. OpenJIT: an open-ended,reflective JIT compiler framework for Java. In *14th European Conference on Object-Oriented Programming*, pages 362–387, 2000.

[25] M. Paleczny, C. Vick, and C. Click. The Java HotSpot^{TM} server compiler. In *Proceedings of the 1st Java^{TM} Virtual Machine Research and Technology Symposium*, 2001.

[26] P. Potrebic. What's the fragile base class (FBC) problem? Be^{TM} newsletter – the newsletter for BeOS^{TM} developers and customers. Issue 79, June 1997.

[27] M. J. Serrano, R. Bordawekar, S. P. Midkiff, and M. Gupta. Quicksilver: a quasi-static compiler for Java. In *Proc. 2000 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 66–82, Oct. 2000.

[28] K. Shudo. shuJIT—JIT compiler for Sun JVM/x86. `http://www.shudo.net/jit/`, 1998.

[29] Sun Microsystems. The Java HotSpot virtual machine white paper. `http://java.sun.com/products/hotspot/`, 2001.

[30] Symbian Ltd. EPOC C++ system documentation – controlling binary compatibility. `http://www.symbian.com/`, 1999.

[31] The FLINT project. Binary compatibility report. `http://flint.cs.yale.edu/~dachuan/bincomp/main.ps.gz`, 2001.

[32] The Free Software Foundation. The GNU compiler for Java. `http://gcc.gnu.org/java/`, 2000.

A Lightweight Java Virtual Machine for a Stack-Based Microprocessor

Mirko Raner
PTSC
(formerly "Patriot Scientific Corporation")
10989 Via Frontera
San Diego, CA 92127
raner@acm.org

Abstract

The large majority of modern JVM implementations are either pure software VMs on top of standard general purpose microprocessors (e.g., Insignia's Jeode or IBM's Jalapeño VM) or Java-specific microprocessors with supportive software layers (e.g., Fujitsu's MB 86799 or aJile's aJ-100).
In this paper a somewhat different approach is presented: a lightweight software VM on top of a general purpose *stack-based* microprocessor. Said microprocessor is not a "hardware JVM", but its architecture is very similar to the JVM. Being truly a general purpose processor, it is not entirely dedicated to execute Java bytecode but can at the same time run C or FORTH applications.

This paper describes the implementation of a lightweight Java Virtual Machine for the IGNITE family of stack-based microprocessors. To achieve an optimal Java performance this JVM uses a combination of standard techniques, such as ahead-of-time (AOT) compilation, class hierarchy analysis (CHA), lazy class loading and binary rewriting, complemented by new optimizations like executable method access structures (XMAS) and lazy argument passing.
The unusual architecture of the target processor also often posed unusual problems that had to be solved.

1 Introduction and Project Background

Originally, the IGNITE microprocessor (formerly named PSC 1000A [PTS00]) was designed as an embedded computing platform for efficient execution of C and FORTH. It has a 32-bit dual-stack architecture (see figure 1) with an 18-word operand stack and a separate 16-word stack for the call stack frames (containing the local variables). Both stacks act as on-chip stack caches and are automatically spilled to and refilled from memory.
The processor uses byte-sized instructions, fitting up to four instructions into one 32-bit machine word. Thus, the IGNITE processor can be considered a real "bytecode" processor (though not a *Java* bytecode processor). Special instruction formats are used for encoding branches and 8-bit or 32-bit constants.
Simplicity was the main design goal of the processor. To save space on the silicon die, it does not have conventional data or instruction caches, and instead of a multi-stage pipeline architecture it uses a simple instruction pre-fetch.

When Java became a popular programming environment, the obvious similarities between the IGNITE microprocessor and the JVM inspired a port of a PersonalJava Application Environment (PJAE). This port was based on Wind River Systems' VxWorks 5.4 as underlying real-time operating system and the Personal JWorks 3.0.2 PJAE ("PJWorks") [Win99]. To take advantage of the IGNITE's architectural similarities to the JVM, a JIT compiler was added to PJWorks and the interpreter loop was recoded in assembly language.

Surprisingly at first, the actual performance of this JVM port fell far behind the calculated estimates. The scores of the CaffeineMark test reached on average only about 40% of the expected scores – even with JIT compilation and a hand-coded interpreter loop! Though the JIT compiler yielded performance gains of up to 10 times over the original interpreter loop, the system was still about 2.5 times slower

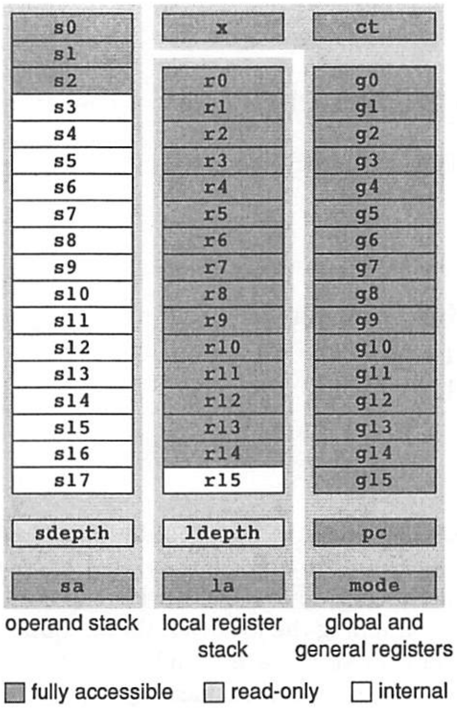


Figure 1: The register set of the IGNITE I

than predicted. A closer examination revealed that the PJWorks design (based on Sun's 1998 version of the JVM) caused a number of severe overheads for the IGNITE processor. Those problems could hardly be overcome without changing the overall architecture completely.
Eventually, the insufficiencies of the PJAE port led to the idea of a new optimized JVM for the IGNITE platform.

Section 2 gives an overview of the new Java architecture for the IGNITE processor; it presents an analysis of the problems of the earlier PJAE port (2.1) and summarizes the design goals of the new system (2.2). Section 3 introduces the low-level OS API used for the new JVM and section 4 covers the implementation details of the new lightweight virtual machine itself. The optimized method invocation scheme of the VM is covered separately in section 5. Finally, section 6 provides preliminary benchmark results and concludes the paper.

2 Overview of the New Architecture

The outcome of the redesign effort, after the disappointing results with the initial PJAE port, was the Lightweight Virtual Machine (LVM) with its low-level API named "JELLO".
For reasons of simplicity and rapid prototype development, the first version of the LVM is based on the reduced instruction set of the CLDC specification – even though the IGNITE processor in fact *does* have floating point instructions.

2.1 Problem Analysis for the Initial PJAE Port

It turned out that the reasons for the low performance of the initial PJAE port were distributed across all layers of the system: the VM implementation, the underlying OS and the microprocessor itself.
The deficiencies were not only located within the components themselves: a lot of the problems were actually caused by the interaction *between* the individual components.

One of the main reasons for the performance limitations was found in the predefined architecture and data structures. Sun's PJAE reference implementation, which is also the core of Wind River's Personal JWorks 3.0.2, was designed for high portability and not necessarily for high performance. It contains a number of abstraction layers which facilitate porting to new platforms but often impede an efficient implementation on a particular processor.
Though a lot of the crucial JVM code was recoded in assembly language there were still large overheads imposed by the interaction with Sun's and Wind River's parts of the VM. Especially jumping between different types of methods (interpreted, JIT-translated, JNI-native, NMI-native, etc.) turned out to be very expensive in some cases (see section 5.4 for more information about "invokers"). In fact, the classical JIT compiler approach is completely inadequate for a target architecture that is so similar to the JVM.

Being originally targeted for the C and FORTH programming languages, the IGNITE processor also uses a different "procedure call standard" than the JVM. That is, it uses a different way of passing arguments to a called procedure (or method). Furthermore, the IGNITE processor uses a different notion of the concept of "stack frames": the call stack (the "local register stack" in IGNITE nomenclature) and the operand stack are separate structures, and the procedure call logic has to establish links between the call stack and the corresponding entries in the operand stack.
Those problems are processor-specific problems.

However, not being limited to the predefined mechanisms and data structures of the reference implementation makes it much easier to find optimized solutions for these problems.

The original PersonalJava VM was mainly written in C and naturally made frequent use of C `structs` to group and handle data. The C language allows structures to be passed as arguments, but the IGNITE architecture is not quite made for that, because the on-chip stack caches have a very limited size and procedure calls get very inefficient when the arguments do not fit into the cache. Therefore it is necessary to use an auxiliary stack, which solves the problem but introduces another considerable overhead.

Finally, VxWorks was also not the best choice for the base operating system. The thread models of Java and VxWorks are fundamentally different and it takes considerable efforts to make one run on the other. In addition to that, the real-time requirements of VxWorks and the non-real-time nature of Java are hard to reconcile, since real-time behavior is hard to guarantee in an environment that depends on automatic garbage collection (though there are approaches that enable real-time garbage collection).
Personal JWorks is also very closely tied to VxWorks, which was another problem, since it made it very hard to change to a different operating system.

Besides the problems of lacking performance and flexibility, the port of Personal JWorks also had an excessively large memory footprint. After JVM-startup about 5 MB of memory had already been consumed, which is a very large amount for a Java solution that is intended to be used in embedded applications.

2.2 Design Goals for the New Java Architecture

After the detailed problem analysis of the earlier JVM port, the following design goals were established for the optimized IGNITE JVM architecture:

1. Increase the Java performance
2. Decrease the memory footprint size of the overall OS/VM combination
3. Take advantage of the unique processor features of the IGNITE
4. Provide the flexibility to use different underlying operating systems
5. Avoid the problems that were caused by choosing C as implementation language

The footprint goal was to have a combined static/-dynamic footprint of about 2 MB for the JVM, including low-level OS functionality required by the JVM.

2.3 Feature Overview

Various different measures were taken in order to meet all the goals stated in section 2.2.
In comparison to the previous Virtual Machine port, the new architecture has four major optimizations:

- Proper separation from the underlying OS
- Lazy class resolution
- Pure Ahead-of-Time compilation without JIT or interpreter
- Optimized method invocation

Probably the most interesting is the optimized method invocation, which is discussed separately in section 5. The optimization features were especially aimed at design goals 1, 2 and 3.
In order to gain more flexibility towards the choice of the underlying operating system (design goal 4), an OS abstraction layer – called "JELLO" – was introduced. JELLO is covered in more detail in section 3.

Finally, in order overcome the implementation problems with C (auxiliary stack for `structs`, different procedure call standards), the LVM was entirely implemented in Java and assembly language. The Java parts (such as the AOT compiler) are converted to IGNITE code via bootstrapping. Using Java as main implementation language has already proven to be a successful idea in other JVM projects, such as the Jalapeño VM [A$^+$00]. The current implementation of the low-level API is still written in C and is based on PTSC's "monitor", which is a minimal operating system for the IGNITE processor.
All JELLO interface functions were specified on machine level (by specifying the operand stack contents before and after a low-level system call).

3 The Low-Level API (JELLO)

The JELLO API (Java Environment Low-Level Operations API) is an API that facilitates the implementation of the Java Virtual Machine on top of an

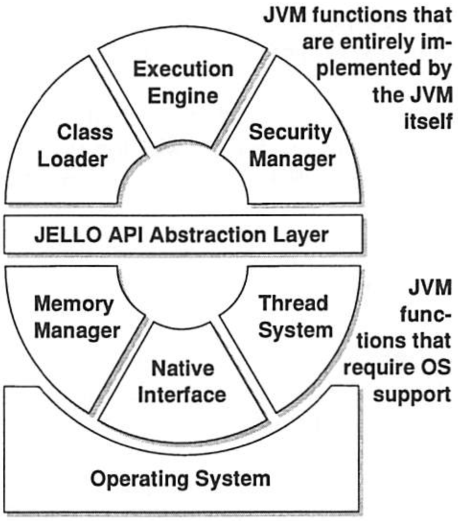

JVM functions that are entirely implemented by the JVM itself
Execution Engine
Class Loader
Security Manager
JELLO API Abstraction Layer
Memory Manager
Thread System
Native Interface
JVM functions that require OS support
Operating System

Figure 2: The JELLO API Layer

underlying operating system that is running on the IGNITE processor architecture.
JELLO is designed for the IGNITE processor but it can also be used in addition to other OS/JVM abstraction layers, if the design of the particular API does make sense in conjunction with JELLO.

3.1 API Design

The design of the JELLO API is based on an analysis of the interaction between the JVM and the OS in the classic JVM model [Ran99]. Whereas execution engine, class loader and security manager can be independently implemented in the JVM, the remaining components – thread scheduler, memory manager and native interface – are directly dependent on OS functionality for their implementation.
Of course, *indirectly* all components depend on OS functionality. However, the functionality can be accessed through lower-level components of the JVM itself. For instance, the class loader will need to allocate memory for new classes and perform file I/O, but in order to do so, it can use the memory manager and native interface of the JVM (through JELLO) – no *direct* access of OS functions is necessary (see figure 2).

Especially when it comes to memory management and thread scheduling, the JVM has very specific demands. Thread prioritizing and context switching has to be handled in a certain way, and memory management has to support automatic garbage collection. However, most operating systems do not support garbage collection and their multithreading API significantly differs from the Java thread model.
The JELLO API encapsulates all the OS functionality that is needed by a JVM and wraps all functions in a way so that they can be directly used for the JVM implementation. For example, the `/setPriority` function of JELLO uses the same priority levels as the `setPriority` method of Java's `java/lang/Thread` class and the memory allocation functions allow passing of additional type information for the garbage collector.
This orientation towards the JVM – as opposed to an orientation towards the OS – distinguishes JELLO from other low-level implementation APIs (such as Sun's Java HPI (Host Programming Interface); [LY97]). In order to change to a different operating system, only the JELLO API layer has to be ported, the JVM on top of JELLO does not require any modifications.

JELLO does not only separate the JVM from the OS but it also isolates the JVM as far as possible from language-, processor- or OS-related idiosyncrasies (such as the auxiliary C stack or a Java-incompatible OS thread model).
However, the JELLO API is specifically for the IGNITE processor and will not make much sense for other processor architectures.

3.2 JELLO Object Layout

JELLO does not know anything about the internal data structures of the JVM. The JELLO API specifies only a small number of data structures that are necessary for the communication between JELLO and the JVM. The most essential data structure specified by JELLO is the layout of JELLO objects in memory (see figure 3; the diagram uses the OVAL [Ran00] notation).
Because JELLO is responsible for memory allocation and automatic garbage collection, the JVM at least needs to convey the size of an object and pointer map information for the garbage collector.

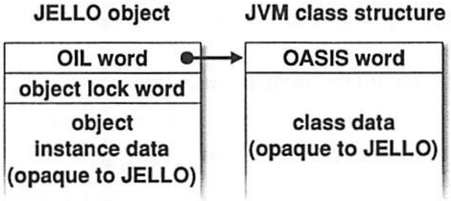

JELLO object
JVM class structure
OIL word
object lock word
object instance data (opaque to JELLO)
OASIS word
class data (opaque to JELLO)

Figure 3: JELLO Object Layout

31	6	1 0
pointer map	size	00
pointer to structure		10
size of primitive array		01
size of object array		11

Table 1: OASIS words for small objects, large objects, primitive arrays and object arrays

This is done by the "OASIS" word – containing the Object Allocation and Storage Information Structure.

Every JELLO object has a 2-word header: the first word is always a pointer (!) to the OASIS word; this pointer is also called the Object Identification Link (OIL). The second word is reserved for the object lock. Currently the second word is just a plain pointer to a monitor structure but more clever mechanisms like inflatable locks or relaxed locks [Dic01] will be used in future versions of the LVM. From the third word of the object onwards, all data is private data, which is opaque to JELLO. The garbage collector knows how big the object is in memory and which of the words after the 2-word header are pointers. All other details of the object layout are entirely up to the JVM implementation.

The OASIS word will usually be the first word of a larger object identification structure. This structure can be the `class_info` or class block – but that does not necessarily have to be so. In fact, the LVM uses this space for the virtual method table (VMT). Slot 0 is reserved for the OASIS word, slot 1 is always used for the `getClass()` method, which gives access to the full class data.

Two objects that have the same OIL pointer value (that is, they point to the same OASIS word in memory) always belong to the same class.

Depending on the class or array that it describes, the OASIS word can have different forms (table 1). For small objects, all required information is stored in the OASIS word itself, for larger and more complex objects the OASIS word turns itself into a pointer. The nonpointer form can be used for all objects that do not have more than 128 bytes of instance data and contain references only in the first 25 words (100 bytes) of the object. These restrictions apply because the OASIS word for small objects uses 5 bit for storing the object size ($4 * 2^5 = 128$) and 25 bit for the pointer map.

Objects that do not qualify as "small", use the pointer form of the OASIS word, which points to one word that contains the object size, followed by one or more pointer map words (one of the 32 bits is used as an "end" marker).

JELLO offers an API for allocating memory (for use by `new`, `newarray` etc.) and explicit GC. It also guarantees that automatic GC is performed in a JVM-compliant manner. However, it does not provide an API specifically for the implementation of GC algorithms (like, e.g., that of the EVM [WG98]). A GC implementation for the LVM must adhere to the JELLO memory layout and provide adequate implementations for the GC-related JELLO functions. It can use any functionality supplied by the OS or by other JELLO components.

4 LVM – The Lightweight Virtual Machine

The main improvements of the LVM itself were already listed in section 2.3: AOT compilation, "lazy" class loading and an optimized method invocation scheme. AOT compilation and lazy class loading are discussed in sections 4.1 and 4.2. Method invocation is comprehensively covered in section 5. Sections 4.3 and 4.4 deal with issues of garbage collection and the native interface of the LVM.

4.1 Ahead-of-Time Compilation

The goal of traditional "just in time" compilation is to find a reasonable compromise between the performance gained by executing compiled methods and the performance lost during the compilation process itself. Usually, only selected methods will be compiled, while others remain interpreted in order to avoid the time penalty for their translation.

This is a sensible approach for most microprocessor architectures, since often very complicated register mapping problems have to be solved in order to compile Java bytecode for a non-stack-based microprocessor.

One disadvantage of JIT compilation is that Java class files have to be kept in memory. Freeing memory used for the bytecode of individual methods within a class file is possible, but is very complicated and can lead to memory fragmentation so that a lot of JIT compilers don't actually do it. Another problem of the JIT approach is the potentially very costly interaction between JIT-compiled and interpreted code (see also sections 2.1, 5.4).

The LVM completely translates a class file immediately after it has been loaded. All information from the class file is retained in the translated version of

the class and the original class file is discarded. All class files are loaded into a single designated temporary buffer.
Translation delays are not an issue for the LVM because most Java bytecode instructions can be mapped one-to-one to corresponding IGNITE instructions or short instruction sequences; the translation process is basically a simple table lookup.

The ahead-of-time (AOT) approach was primarily chosen to avoid the previously mentioned disadvantages of JIT compilation. Also, the compromise between translation time penalties and performance gains is not necessary on the IGNITE platform, as the translation time is negligible in comparison to the loading time for a class file.

4.2 Lazy Class Loading

Class loading and resolution can be performed in a "static" or in a "lazy" manner [LY99, §2.17.3]. With a few limitations, a JVM implementation can freely decide when a particular class is loaded and resolved.
For example, a simple test can be used to reveal the class loading policy of Sun's Java 2 VM. On a UNIX system, using the command line

```
java -verbose:class HelloWorld|grep Loaded|wc -l
```

one can determine how many classes are loaded for a simple `HelloWorld` program.[*] A surprising class count between 160 and 210 can be observed for most Sun Java 2 VMs – for a simple `HelloWorld`.
In some JVM implementations loading of one particular class can spawn a whole tree of other classes that are loaded.

The LVM defers the resolution of classes and loading of additional classes as much as possible. The minimum requirement is that the superclass of every newly loaded class must be loaded and resolved before that class. This is necessary in order to properly construct the virtual method tables and the memory layout of objects. Besides that, the LVM always leaves external references unresolved until the moment they are actually used for the first time. So, for example, instead of loading an exception class just because some method declares that it might throw that type of exception at some point in time, the loading of the exception class is delayed until that particular exception is actually thrown (or, more precise, until an exception object is created).

The method invocation scheme of the LVM allows that references to classes, methods and fields can stay unresolved until they are accessed; for details see section 5.

4.3 Garbage Collection Algorithms

Currently, the LVM uses a simple, merely conservative mark & sweep garbage collector. The JELLO object memory contains accurate pointer maps for all objects, however, the stack frames on the call stack do not have pointer maps yet. Later versions of the LVM might provide pointer maps for the stack frames as well.
With the exception of hybrid garbage collectors that also do reference counting, most common garbage collection algorithms can be used with the LVM. Reference counting is problematic because additional reference adjusting code would need to be added and the one-to-one translation between Java bytecode and IGNITE code would be lost. For example, `dup` can be translated to the IGNITE instruction `push s0`; if reference counters are involved, this simple translation will no longer work, because the reference count of the top-of-stack object needs to be adjusted as well.

Section 3.2 contains further details on memory layout and garbage collection in the LVM.

4.4 Native Methods

The LVM is modeled after the CLDC specification and therefore it does not need to implement the Java Native Interface (JNI).
Still the LVM has to provide a way to implement methods that cannot be written in Java because they need to access low-level OS functionality.
In addition to this basic native method problem, there is also the question how the LVM classes themselves can access low-level functionality such as reading and writing memory. The Jalapeño VM, for example, solves the latter problem by its MAGIC class [A$^+$00, appendix A].

In the LVM both problems are solved with the same technique. Similar to the `asm` directive of many C and C++ compilers, the LVM offers an IGNITE inline assembly directive. Table 2 shows an example of IGNITE native programming.[†] Java source code

[*] an additional `2>&1` might be necessary for some JVMs that print verbose messages on `stderr` instead of `stdout`.

[†] In fact, the listing does not show the actual implementation of the hashcode function; the shown implementation was chosen to demonstrate the use of labels and branches in inlined native code; the real implementation simply uses 3 `SHR_1` instructions instead of a complicated loop.

```
package com.ptsc.lvm.java.lang;
import com.ptsc.lvm.java.Native;
public class System implements Native
{
    /** Returns object address >> 3. **/
    public static int
    identityHashCode(Object object)
    {
        int[] $native =
        {
            PUSH_S0,        // clear carry
            AND,
            PUSHN_3,        // do 3 shifts
            POP_CT,         // set loop ct
        ~1, SHR_1,          // shift
            DBR, ~1,        // loop back
            RET
        };
        return $INT;
    }

    /** Explicitly trigger a GC. **/
    public static void gc()
    {
        int[] $native =
        {
            BR, $._("/gc")// call JELLO
        };
        return;
    }
...
```

Table 2: An example for LVM native programming

that contains inlined assembly code can still be compiled with any standard Java compiler (though it looks very unusual at first). When the class files for such native classes are executed on a standard JVM, they will produce an error – or simply do nothing in the best case. However, the AOT compiler of the LVM will recognize the special content of those classes and produce the equivalent IGNITE machine code. In fact this technique is pretty common and is used in Jalapeño as well as in Jbed [TMH99].

The `Native` interface contains integer constants for all IGNITE instructions. The spelling of the instruction mnemonics has been slightly adapted, so that the constant identifiers conform to the Java syntax and the Java naming conventions [GJSB00, §6.8.5]. For instance, `push s0` turns into `PUSH_S0` and `br []` becomes `BR_$$`.
Besides the instruction mnemonics, the `Native` interface also provides constants that indicate the return type of a method (e.g., `$INT` or `$BYTE`), a general label symbol (`LABEL-` or `~`) and a symbolic reference operator (`$`).
Labels are defined and referred as `~0`, `~1`, etc. or `LABEL-1`, `LABEL-2` etc. JELLO API functions or symbolic method references can be located with `$._("`⟨*method name*⟩`")`.

The class names of the JDK library classes for the LVM are prepended with `com.ptsc.lvm` so that they can be compiled with standard Java compilers. Again, the LVM AOT compiler will convert the package specifications back to the original JDK package names.

5 Optimized Method Invocation

Method invocation is one of the most crucial operations in an object-oriented system. If method invocation is slow or subject to unnecessary overheads, this will also degrade the overall system performance.
For this reason, method invocation in the LVM was optimized with the following techniques:

- Devirtualization based on Class Hierarchy Analysis (CHA)
- Executable Method Access Structures (XMAS)
- Binary rewriting
- Lazy argument passing

Those optimizations, and how they apply for the LVM, are described in more detail in sections 5.3 to 5.8. Section 5.1 describes in general how Java-style method invocation can be implemented on the IGNITE platform and section 5.2 deals with IGNITE-specific stack frame problems.

5.1 Phases of Method Invocation

Procedure calls on the IGNITE dual-stack architecture and method invocations in the JVM are substantially different.
The Java Virtual Machine passes arguments on the operand stack, but for the called method the arguments appear in the local variables (which correspond to the IGNITE's local register stack). On the IGNITE the arguments stay on the operand stack and need to be moved to the right place so that the computation model is not violated.
The `invoke...` instructions of the JVM also perform a lot of additional functions, besides merely transferring execution to another place. They take care

	Operation	Comments
1	Resolve method	needs only be done once for previously unresolved methods
2	Check whether the object is `null`	not necessary for `static` methods; can be omitted if object is "`this`"
3	Find correct method code	only necessary for true virtual methods; can be optimized in many cases
4	Transfer arguments to local register stack	can be optimized with lazy argument passing
5	Transfer execution to target method	
6	Create register stack space for variables	
7	Establish operand stack link for stack frame	

Table 3: Phases of method invocation

of method resolution, check whether the caller is using a `null` reference, automatically dispatch virtual methods and allocate a new stack frame for the called method. All these things have to be done "manually" on the IGNITE processor.

Table 3 summarizes the phases of method invocation.

5.2 Linking the Call Stack to the Operand Stack

In the computation model of the JVM every stack frame on the caller stack actually *contains* the operand stack for the method. Since the required size of the operand stack is a known constant [LY99, §4.7.3] this does not cause any problem. Stack frames can be implemented such that the operand stack part of one stack frame overlaps with the local variables of the next frame. Overlapping stack frames completely eliminate the need for copying arguments to the local variables of the receiving method and can be implemented very efficiently on a large number of architectures (including SPARC).

Unfortunately, this is not true for the dual-stack architecture of the IGNITE processor.

Operand stack (`s0` – `s17`) and call stack (`r0` – `r15`) are separate structures in the IGNITE architecture. The operand stack is used for all computations, the call stack only stores variables and the return address.

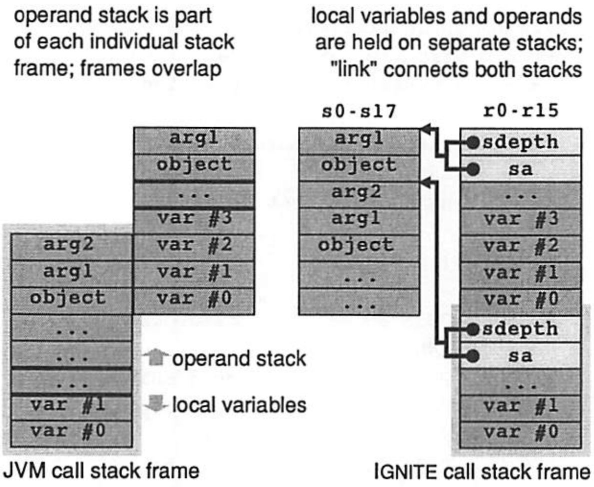


Figure 4: JVM stack frames versus IGNITE stack frames (simplified)

There are two potential ways to simulate a JVM-style stack structure on the IGNITE:

- Save the whole contents of the operand stack into the local register stack (not only the arguments but also the data that might be on the stack *below* the arguments)
- Leave the residual contents of the operand stack where they are and establish a link between the stack frame on the local register stack and the "operand stack frame" on the operand stack; in case of an exception the link allows a rollback to the correct operand stack state

Clearly, the latter solution is the favorable one, because it involves less data shifting between the two stacks.

Unfortunately, it is not possible to find out the projected memory address of an operand stack element directly. The processor only offers the current base stack pointer for the operand stack (`sa`) and the number of elements currently in the on-chip cache (`sdepth`). The projected address of a stack element in memory can be calculated from those two values. To save time, both `sa` and `sdepth` are stored in the local register stack to form the operand stack link. Only in case of a rollback (that is, an exception) the effective address is actually calculated.

Figure 4 visualizes how overlapping Java stack frames are simulated on a dual-stack architecture.

5.3 Devirtualization based on CHA

Virtual method invocations can often be replaced by faster nonvirtual invocations, which do not require a VMT lookup [DGC95].
Except for constructor calls and invocations of `private` or `super` methods, Java treats all non-static method invocations as virtual. With static and dynamic class hierarchy analysis (CHA) virtual invocations can be converted ("devirtualized") to nonvirtual ones in many cases. This technique is already used by several JVM implementations (e.g., the Harissa VM [MMBC97]).

Java bytecode represents a method reference as a pair of a class reference and a method signature (e.g., `java/lang/Math/round(D)L` or `A/one()V`) [LY99, §4.4.2]. During a virtual method invocation the reference resolves to a particular method implementation, which either belongs to the referred class itself or to a subclass of that class. The referred class might have inherited the method implementation from a direct or indirect superclass, but generally an `invokevirtual` instruction cannot access method implementations of superclasses (`invokespecial` needs to be used in those cases). Together with the loading order of classes used by lazy class loading, this fact can be used for implementing a very simple devirtualization scheme.

The lazy class loader always processes superclasses first, because otherwise it could not set up the virtual method table (VMT) for a class.
Therefore, when a loading request for a particular class is issued, it can be safely assumed that no subclasses of that particular class are present in the system yet (otherwise the requested class would already have been loaded as a superclass, and the loading request would not have been issued in the first place). The LVM takes advantage of this and initially marks *all* methods of a newly loaded class as *devirtualized*. Only, when that class is subclassed and methods are overridden later (or even during the same loading request to the class loader) those methods are reverted to their regular virtual state.

Figure 5 shows a simple class hierarchy (`C extends B extends A`): class `A` defines two methods `one()V` and `two()V`, class `B` overrides `two()V` and adds `three()V`, class `C` again overrides `one()V` and `three()V`.

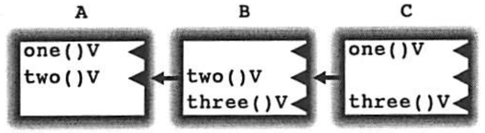


Figure 5: Example class hierarchy (OVAL notation)

class `A` loaded	class `B` loaded	class `C` loaded
◆`A/one()V`	◆`A/one()V`	◇`A/one()V`
◆`A/two()V`	◇`A/two()V`	◇`A/two()V`
	◆`B/one()V`	◇`B/one()V`
	◆`B/two()V`	◆`B/two()V`
	◆`B/three()V`	◇`B/three()V`
		◆`C/one()V`
		◆`C/two()V`
		◆`C/three()V`

◆= devirtualized, ◇= revirtualized

Table 4: Devirtualization of methods

Table 4 shows how the devirtualization status of the methods changes when classes `B` and `C` are added later. It is noteworthy, that, even though some methods are basically the same (e.g., `B/one()V` inherits the code from `A/one()V`), it is necessary to distinguish whether a method invocation refers to `A/one()V` or `B/one()V`.

Sections 5.4 and 5.6 explain in more details how the devirtualization and revirtualization is handled in the LVM.

5.4 Executable Method Access Structures (XMAS)

Every JVM implementation requires data structures that store information about each loaded method. Amongst the stored pieces of information are the address of the method code, the method flags and possibly also the slot offset in the VMT (Virtual Method Table), if such a structure is used. In Sun's JVM terminology those structures are called `method_info` [LY99, §4.6] or "method blocks" [Yel96].
In his document "The JIT Compiler API" Frank Yellin also introduced the concept of "invokers" [Yel96, §6.5]. An invoker is a piece of code that acts as a kind of adapter when execution is transferred between methods of different types. Each method type has a particular invoker associated with it. There are different invokers for interpreted, synchronized interpreted, native, JIT-compiled and other types of methods. Thus, the JVM does not need to know how to call code that has been produced by a particular JIT compiler plug-in; it only has to call the appropriate invoker, which will handle the

	Unresolved	Devirtualized	Revirtualized
`0x00`	`br 0x08`	`call makeshortcut`	VMT dispatcher code (see table 6)
`0x04`		address of method code	
`0x08`	`push.l / push.l`	VMT offset of method	
`0x0C`	pointer to class name	pointer to class structure	pointer to class structure
`0x10`	pointer to signature pair	pointer to signature pair	pointer to signature pair
`0x14`	`br resolve`	access flags	access flags

Table 5: XMAS layout for unresolved, devirtualized and revirtualized methods

interaction.
On the back side, invokers are an additional level of indirection and impose a certain overhead. This overhead adds to any already existing overhead, for example, that of a VMT lookup.

In the LVM the concepts of "method block" and "invoker" are combined into a new data structure which has been given the name "executable method access structure" (nicely abbreviated XMAS).

As the name already suggests, the XMAS contains executable code. Normally, the method dispatching code has to extract the data from the method block in order to transfer execution to the appropriate address.
The LVM uses a slightly different technique: instead of extracting various pieces of information from the data structure and acting accordingly, a method is invoked by simply calling the XMAS like an executable routine. The code in the XMAS *is* already the dispatcher and will transfer execution to the correct address.

Whereas the pointer to the XMAS of a method will always remain the same, the layout and contents of the XMAS will change as the method goes through different stages of its lifecycle. There are different variations of XMAS blocks for unresolved, devirtualized and revirtualized methods. Table 5 shows the different XMAS layouts for methods called by `invokevirtual`.

`push` sp	`ld []`	`push.n #`i	`add`
`br []`			
(#i, repeated)			

(a) Dispatcher code for slot offsets 4 and 8.

`push` sp	`ld []`	`push.b #`i	(#i)
`add`	`br []`		
(#i, repeated)			

(b) Dispatcher code for slot offsets 12 to 252.

`push` sp	`ld []`	`push.l #`i	`add`
(#i)			
`br []`			

(c) Dispatcher code for slot offsets 256 and higher.

Table 6: (a) – (c) Dispatcher code sequences.

An unresolved method is a method whose class has not been loaded yet. Since the LVM uses lazy class loading, this situation may appear very frequently. The XMAS for unresolved methods contains a pointer to the class name and a pointer to the signature pair. When the XMAS is executed those two values are pushed onto the stack and the resolution routine (`resolve`) is called. To fully understand the XMAS layout for unresolved methods, one must know that the IGNITE processor allocates a separate word for each `push.l` constant within an instruction group; the constants follow immediately after the instruction group and are automatically skipped by the decoding logic.

Whenever an unresolved method is actually invoked for the first time, the resolver will load the appropriate classes and will change the XMAS to the "devirtualized" state (see also section 5.3).

If a method gets overridden by another method that got newly loaded, the devirtualization will get undone ("revirtualization"), so as to avoid that falsely optimized methods are called.

For revirtualized methods, which consequently require a VMT lookup, the three first words of the XMAS contain the method dispatcher code. Depending on the VMT slot number of the method there are three different formats how those three words are used (see table 6).
As the LVM uses the JELLO Object Identification Link already as the VMT pointer, slot offset 0 will never be used, because slot 0 contains the OASIS word for the class. Slot offset 4 always branches to the `getClass()` method, which gives access to all other details of a loaded class (see also section 3.2). As a further optimization, slot offset 8 is reserved for the `hashCode()I` method.

The LVM uses the VMT data structure in a slightly

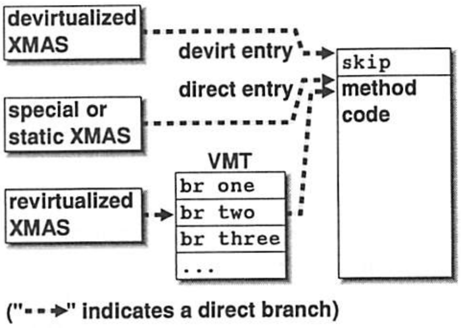


Figure 6: Relations between XMAS and VMT.

different manner compared to other object oriented systems.

Instead of reading the VMT slot contents (instruction `ld []`) and then branching (`br []`), the dispatcher code directly uses `br []`. This is possible because the VMT does not contain the address of the method code but a direct branch to the code (see figure 6). This eliminates one instruction from the dispatcher code and saves space in the XMAS; however from a memory access point of view there is no difference (instead of the `ld []` in the XMAS the `br` in the VMT requires an additional memory access for fetching the branch target).

The 4-byte version of the `br` instruction can perform relative branches between -268435456 and +268435452, which effectively limits the LVM to a 256 MB contiguous address space. Since the LVM is mainly targeted for small embedded devices, this does not pose a real limitation. Another side effect of the unusual VMT usage is that the VMT of a class can no longer be created by copying the VMT of its superclass and changing and adding some slots. Since the `br` instructions are *relative* branches, the VMT contents also need to be relocated.

There is one important difference between XMAS blocks and the classic JVM method block: for one and the same method implementation multiple XMAS blocks might exist, because an inherited method implementation is referred to as a member of different classes (see example in section 5.3: method `B/one()V` can be invoked as `B/one()V` but it can also be invoked as `A/one()V` when an object of class `B` is manipulated through a reference of type `A`).

All information that is not stored in the XMAS block is, in fact, appended to the method code. A "magic" number marks the end of the actual code and the beginning of the method's stack frame descriptor, exception table, line number table and other attributes.

Similar to the JTOC data structure of Jalapeño [A+00], the LVM has one hashtable that contains pointers to the class structures of all loaded classes, using the fully-qualified class names as keys.

Each of the class structures has another hashtable containing XMAS pointers for all methods of that class. The name-and-type pairs [LY99, §2.10.2, 4.4.2] of the methods serve as keys for those subtables.

The hashtables are only accessed when classes are loaded or methods are resolved, for example, to change a particular XMAS from unresolved to devirtualized state. For regular method invocations of any type the hashtables do not need to be accessed.

5.5 XMAS for Fields and Static, Special or Interface Invocations

Not only `virtual` method invocations are translated into a call to an XMAS, but also `special`, `static` and `interface` invocations. Even the access to static and nonstatic fields is handled through an XMAS.

Static and special invocations (JVM instructions `invokestatic` and `invokespecial`) use the same XMAS layout as virtual invocations for the "unresolved" and "devirtualized" stages. The difference is that they remain in that state and never get revirtualized. Also, they use different entry points into the method body (figure 6), so that they are not affected by revirtualization that happens through binary rewriting (see section 5.6).

The `invokespecial` opcode is used for calling constructors, private methods and methods of the direct superclass [LY99, §6].

When referring to `private` or `super` methods, or constructors of the superclass or the class itself, `invokespecial` does not use an XMAS. As the superclass of a class always has to be loaded and resolved first (see 4.2), those invocations can be translated into direct calls because their target address is already known at that time. Only invocations of constructors whose classes have not been loaded yet are indirected through an XMAS.

Interface method invocations (`invokeinterface`) use the same format as all other invocation types in their unresolved state. When they get resolved, the XMAS contains code that branches to a special interface method dispatcher. This dispatcher looks at the actual type of the object on which the method is being invoked. The dispatcher then uses a lookup

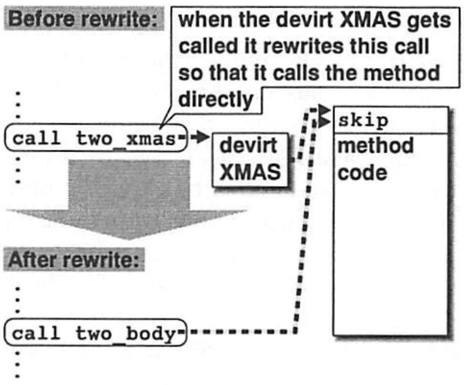

Before rewrite:
when the devirt XMAS gets called it rewrites this call so that it calls the method directly
call two_xmas
devirt XMAS
skip
method code
After rewrite:
call two_body

Figure 7: Turning an indirect call (through the XMAS) into a direct call with binary rewriting.

table to retrieve the interface method table (IMT) for that particular object type. After that, the IMT is used similarly to a VMT.

The JVM instructions `getfield`/`putfield` and `getstatic`/`putstatic` are also translated with the help of an XMAS. In the unresolved state the XMAS forces loading and resolution of the affected class. After resolution, the XMAS leaves the absolute address of the field on top of the operand stack. Read access is simply done with an `ld []` instruction, write access with `st []` and `pop`.

5.6 Binary Rewriting

Binary rewriting is a form of self-modifying code and is known as an optimization technique for a long time. Even so, it is no longer used very often. The reason for this is that most modern microprocessor architectures have large instruction caches, which are very problematic in conjunction with binary rewriting. For example, if self-modifying code is used on a StrongARM SA-1110 processor, a cache synchronization is required after the modification took place. The penalty for such a cache synchronization operation can be in the order of 10000 instruction cycles.

Being a very cheap and simple microprocessor, the IGNITE's only instruction cache is the 4-byte "cache" of its instruction pre-fetch. There is no other instruction cache and therefore there is no synchronization problem when binary rewriting is used.

One of the ideas behind the design of the LVM was that, if a Java method can theoretically be invoked with one single direct branch, the LVM should be able to perform the method invocation with a single branch.
In other words, if the detour through the XMAS turns out to be unnecessary the LVM should use a direct branch.

The XMAS for devirtualized methods calls a routine named `makeshortcut`. This routine transfers execution to the address that is stored in the second word of the XMAS.[‡] Before actually jumping there, it modifies the original `call` instruction in the calling method, so that it now calls the revirtualized ("revirt") entry of the target method directly. Figure 7 visualizes this process.

When the runtime system detects that a devirtualized method must be revirtualized, two things will happen: first, the XMAS of that method will be changed into the "revirtualized" form and, second, the first word of the method code (which originally just contained a `skip` instruction) gets rewritten with a call to a revirtualizer routine ("revirt code"). Any call sites that enter through that first entry point will also get rewritten to the indirect form which branches to the XMAS instead of directly going to the method body. Figure 8 visualizes these steps.

5.7 Lazy Argument Passing

The differences between the argument passing mechanisms of the JVM and the IGNITE make it necessary to move method arguments from the IGNITE operand stack to the IGNITE local variable stack.

However, it can be observed that a lot of Java methods start with the instruction `aload_0` or a sequence like `aload_0`, `iload_1` (or even `aload_0`, `iload_1`, `iload_2`).
Especially, the `aload_0` at the beginning of a method is very common, since this instruction is the bytecode equivalent of a "`this`" reference.

When a method begins with an instruction sequence that fits into this scheme, it effectively just moves data back to where it was before. Thus, during argument passing, the arguments do not need to be moved to the local register stack in the first place and the instructions at the beginning of the method can be optimized away.

One important prerequisite for this optimization is,

[‡] because the routine is called with `call` instead of `br`, the address of the subsequent word is stored in `r0` as actual return address; `makeshortcut` removes that address from the stack afterwards

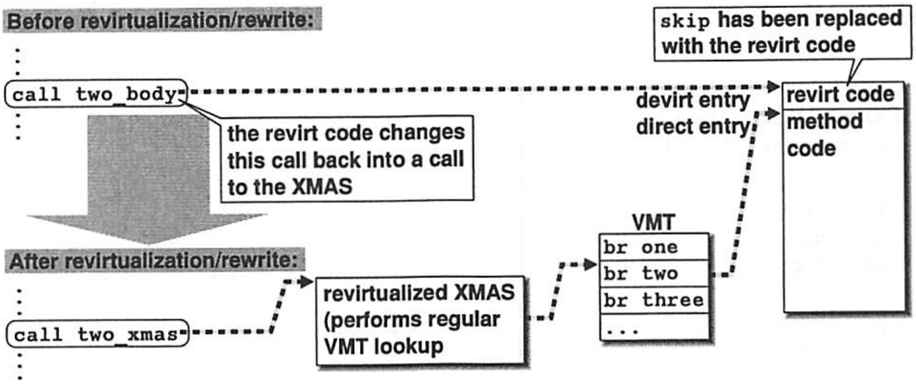

Before revirtualization/rewrite:
call two_body
the revirt code changes this call back into a call to the XMAS
After revirtualization/rewrite:
call two_xmas
revirtualized XMAS (performs regular VMT lookup
VMT
br one
br two
br three
...
devirt entry
direct entry
skip has been replaced with the revirt code
revirt code
method code

Figure 8: Revirtualization process for methods that were using a "shortcut" created with binary rewriting

Vars optimizable:	0	1	2	3
static				
PJava 3.0.2 classes	66.74	24.43	6.45	2.38
KVM Ref. Impl.	69.03	24.73	5.41	0.83
Kaffe `klasses.jar`	58.58	30.20	9.04	2.19
SPECjvm98	71.54	23.07	3.74	1.66
dynamic				
SPEC 200check	60.81	38.53	0.53	0.14
SPEC 201compress	60.16	39.64	0.15	0.05
SPEC 202jess	61.39	37.89	0.58	0.15
SPEC 227mtrt	31.09	65.82	0.74	2.36

Table 7: Optimizable Methods (static and dynamic)

that the local variables which are optimized are not used elsewhere in the method. So, in summary, the criterium is:

- The method starts with *x*load_0 [,*x*load_1 [,*x*load_2]] (in that order)
- There is no other read access of the local variable(s) 0 [,1 [,2]] except at the beginning of the method
- Instruction 0 [,1 [,2]] is not the branch target of a `goto`, `if`*cond* etc.

Already in the static analysis of the classes.zip file for PJava and the KVM reference implementation (table 7, line 1 and 2), it becomes obvious that optimizations of three variables can only be done in very rare cases. However, for about 9% of the methods found in the klasses.jar of the KaffeVM an optimization of 2 variables is possible.

Unfortunately, the dynamic view (table 7, lower part) looks different. The dynamic results for various SPEC jvm98 benchmarks were weighted with the number of invocations that actually happened at runtime. The tests yielded that only optimizations of one variable will make a significant difference. However, depending on the benchmark, between 38 and 65 (!) % of the method invocations benefit from an optimization. However, the actual degree of optimization varies greatly between different types of applications.

The LVM uses a special argument passing scheme (called "lazy argument passing") to apply the above optimizations. For methods that have 3 or less arguments (`this` counts as an argument, too) all arguments are left on the IGNITE operand stack. It is the responsibility of the target method to move the arguments to the local registers - or leave some of them on the operand stack, if the optimization can safely be applied. The LVM AOT compiler can statically decide how many variables can be optimized, if any at all. If possible, it will produce code that applies the optimizations.

If a method has more than 3 arguments, the calling method moves arguments 4 to *n* to the local register stack. The first 3 arguments are left on the operand stack.

The above scheme has another inherent advantage: The IGNITE microprocessor can only access the top three operand stack registers (`s0`, `s1`, `s2`). In the original JVM method invocation scheme, the object reference, which determines which method is the correct receiver of an `invokevirtual`, is at the bottom of the operand stack - buried deep below all the additional arguments. With the LVM lazy argument passing scheme, the object reference can easily be copied to the top of stack by issueing a `push s`*p* (where *p* is 0, 1, 2; see also table 6).

5.8 Translation Example

A simple example can illustrate how the AOT compiler of the LVM translates method invocations.
The example shows the translation of the write([CII)V method in class java/io/BufferedWriter. The method has 4 arguments (the implicit this argument and 3 additional arguments). An invokevirtual of this method is translated as follows:

```
pop lstack        ; move 4th argument to LR stack
push s2           ; get object reference
skipnz            ; check object against null and
call /throw_s2 ; trigger exception if necessary
call xmas...      ; execute XMAS of method
```

It is the responsibility of the called method to establish the stack link, move the first three arguments to the local register stack (if necessary) and allocate space for additional local variables (if necessary).

6 Conclusion

Currently, the LVM is still lacking most of the standard CLDC class libraries, which is the reason why only a limited number of benchmark results for the outdated Embedded CaffeineMark suite is available. In comparison to the earlier PJAE port, the score in the ECM3 "Method" test was increased from 50 to 138. The "Logic" score was 187 instead of 96, and the "Sieve" score went up from 128 to 205 (all results were measured on a 100 MHz IGNITE NC1 reference board). Even though the LVM is still far away from a complete JVM/KVM product, it proved that especially the optimized method invocation scheme is advantageous to the IGNITE architecture.

The IGNITE I processor is a "softcore" described in VHDL, which means that new instructions can be added and tested easily. A netlist for Xilinx Virtex FPGAs and other common FPGA devices can be freely downloaded from the PTSC website (http://www.ptsc.com) for evaluation.
Possible near future developments of the processor are additional instructions for null pointer testing, stack frame linking and additional loop instructions (for ifle, ifgt etc.).

References

[A$^+$00] B. ALPERN / OTHERS. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211 – 238, 2000.

[DGC95] J. DEAN / D. GROVE / C. CHAMBERS. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the ECOOP'95 Conference*, 1995.

[Dic01] D. DICE. Implementing Fast Java Monitors with Relaxed-Locks. In *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM'01)*, pages 79 – 90, Monterey, CA, 2001.

[GJSB00] J. GOSLING / B. JOY / G. STEELE / G. BRACHA. *The Java Language Specification.* Addison-Wesley, Reading (MA), 2nd edition, 2000.

[LY97] T. LINDHOLM / F. YELLIN. Java Runtime Internals. Presented at the JavaOne'97 Conference, Track 1, Session 27. San Francisco (CA), 1997.

[LY99] T. LINDHOLM / F. YELLIN. *The Java Virtual Machine Specification.* Addison-Wesley, Reading (MA), 2nd edition, 1999.

[MMBC97] G. MULLER / B. MOURA / F. BELLARD / C. CONSEL. Harissa: a Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems (COOTS)*, Portland (OR), 1997.

[PTS00] PTSC, San Diego, CA. *PSC1000A Microprocessor Reference Manual*, August 2000.

[Ran99] M. RANER. Implementation der Java Virtual Machine (*in German only*). *Java Magazin (Germany)*, pages 34 – 40, issue #6.99, 1999.

[Ran00] M. RANER. Teaching Object Orientation with the Object Visualization and Annotation Language (OVAL). In *Proceedings of the ACM ITiCSE 2000 Conference*, pages 45 – 48, Helsinki, Finland, 2000.

[TMH99] J. TRYGGVESSON / T. MATTSSON / H. HEEB. Jbed: Java for Real-Time Systems. *Dr. Dobbs Journal*, 24(305):78 – 86, November 1999.

[WG98] D. WHITE / A. GARTHWAITE. The GC Interface in the EVM. Technical Report SML TR-68-67, Sun Microsystems Laboratories, 1998.

[Win99] WIND RIVER SYSTEMS, Alameda, CA. *Personal JWorks Programmer's Guide 3.0*, 1st edition, May 1999.

[Yel96] F. YELLIN. The JIT Compiler API. URL: http://java.sun.com/docs/jit_interface.html, 1996.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits

- Free subscription to *;login:*, the Association's magazine, published eight times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and Open Source, book and software reviews, summaries of sessions at USENIX conferences, and Standards Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to *;login:* on the USENIX Web site.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, on the USENIX Web site.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as security, Linux, Internet technologies and systems, operating systems, and Windows—as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Savings on a variety of products, books, software, and periodicals: see *http://www.usenix.org/membership/specialdisc.html* for details.

Supporting Members of the USENIX Association

Freshwater Software
Interhack Corporation
Lucent Technologies
Microsoft Research
Motorola Australia Software Centre
OSDN
The SANS Institute
Sendmail, Inc.
Smart Storage, Inc.
Sun Microsystems, Inc.
Sybase, Inc.
Taos: The Sys Admin Company
TechTarget.com
UUNET Technologies, Inc.
Ximian, Inc.

Supporting Members of SAGE

Certainty Solutions
Collective Technologies
ESM Services, Inc.
Freshwater Software
Lessing & Partner
Microsoft Research
Motorola Australia Software Centre
New Riders Press
O'Reilly & Associates Inc.
OSDN
Ripe NCC
Taos: The Sys Admin Company
Unix Guru Universe

For more information about membership, conferences, or publications,
see *http://www.usenix.org/*
or contact:
USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA
Phone: 510-528-8649 Fax: 510-548-5738 Email: *office@usenix.org*

ISBN 1-931971-01-3